A User-centric approach to Resilience

Alexander Schwind Zied Baatour

***Abstract*—This report tackles the problem of fault tolerance and resilience within microservice architectures using the TeaStore application as a case study. The original system shows tight coupling and single points of failure that leds to cascading failures severely impacting core functionalities. To address this, we implemented a refactoring effort centered on a user-centric approach to resilience and fault tolerance. Key architectural changes included decoupling the monolithic Persistence service into multiple domain-specific services and integrating the Temporal workflow engine to ensure the durable execution of stateful business processes. We implemented several resilience patterns, including replication, caching, and fallbacks, motivated by the goal of preserving the end-user experience. The success of our refactoring was evaluated by measuring the number of available features during targeted service failures. The results shows a substantial improvement in system resilience and fault tolerance and we are able to generalize some of the refactoring efforts to other applications. The application is able to degrade gracefully and maintain critical functionality, thereby validating our approach as an effective strategy for building a fault-tolerant, resilient cloud-native system.**

I. Introduction

In distributed systems, the terms fault, failure, and error are often used interchangeably, but a precise vocabulary is essential for a focused approach. For this report, we adopt the following definitions. A fault [1] is a defect in a component. When that fault is activated, it can lead to an error [1], which is an incorrect internal state. A failure [1] occurs when a system deviates from its intended function or specification. Our goal is to prevent failures from happening or to ensure a degraded but still usable functionality. This leads to two key system properties:

- Fault Tolerance: The system's ability to continue functioning like specified, even when some components fail [4].
- Resilience: The ability of a system to recover gracefully from failures. This includes not only withstanding the initial shock but also adapting and returning to a fully functional state once the fault is resolved [3].

In distributed environments where the failure of a single node can have wider repercussions, a fault tolerance strategy is essential [2, 5]. Such a system should automatically detect, isolate, and manage faults to maintain normal operations. This project confronts the challenge of improving both fault tolerance and resilience within TeaStore, a microservice application. Our initial analysis revealed a concrete and critical problem scenario that guided the refactoring effort: a failure in the monolithic teastore-Persistence service caused a catastrophic, cascading failure across the entire application, rendering almost all features unusable. This single point of failure meant that a transient fault in one database-dependent service could halt the entire user journey causing an unacceptable user experience.

The primary goal of this project was therefore to solve this problem by transforming the TeaStore from a tightly coupled system into a resilient one that degrades gracefully. We aimed to refactor the architecture so that the failure of any one single microservice would have a minimal and localized impact.

Here we only consider the fault cases where one service becomes unavailable from the point of view of other services. In our controlled experiments we kill the running pod, but in production environments a service can become unavailable due to multiple reasons like network partitioning, node failure etc.

To measure our success, we adopted a user-centric methodology, defining resilience by the tangible functionality available to an end-user during a partial outage.

This report details our journey in resolving the identified problem. We describe the architectural transformation, centered on decoupling and separating the databases and introducing a durable workflow engine (Temporal) to manage stateful logic. We then explain the implementation of specific resilience patterns, such as caching and fallbacks, for each service. Finally, we present the evaluation of our results, using our feature availability metric to provide clear and convincing evidence that we successfully achieved our refactoring goal.

II. Our Approach: A User-Centric Methodology

To address the shortcomings of the baseline TeaStore, we adopted a methodology that prioritizes the end-user's experience. Although metrics such as uptime or mean time between failures are useful to assess reliability, the probability that the system performs its intended function correctly over time [3], they could fail to capture the nuanced reality of a partial system failure and are also not realistic in our setup. A system can be technically up but functionally useless if a critical dependency is unavailable. Our approach, therefore, was designed to quantify resilience and fault tolerance in terms of tangible, user-facing functionality. This methodology consists of four stages.

1) First, we defined the expected user experience by decomposing the application into its essential user-facing features. This resulted in a specification of 12 core functionalities that represent a complete and successful user journey, from initial product discovery to checkout. These functionalities, detailed in Table **??**, served as the basis for our evaluation.

2) Second, we systematically tracked violations of this expected behavior in the baseline architecture using chaos engineering. This analysis, detailed in Section 3, allowed us to pinpoint which architectural weaknesses were responsible for the most severe impacts on the user.
3) Third, based on this analysis, we identified and implemented common microservice patterns that tackle the revealed issues. Our refactoring strategy focused on decoupling critical services, isolating failure domains, and implementing graceful degradation patterns such as fallbacks and caching.
4) Finally, we compared the before and after states by reapplying the same chaos experiments to our refactored architecture. This allowed for a direct, qualitative and quantitative comparison of the system's fault tolerance and resilience, measured by the number of core functionalities that remained available during a given service failure.

Category	**Functionality**
Discovery	1. See category list 2. See products in a category 3. See product information 4. See product images 5. See recommendations
Shopping Cart	6. Add items to cart 7. Update item quantity 8. Remove item from cart
Account	9. Login and Logout 10. See profile information 11. See past orders
Checkout	12. Order cart items

TABLE I: The 12 Core User Functionalities of the TeaStore

III. Analysis of the Baseline Architecture

Following the methodology outlined above, we performed an analysis of the TeaStore to establish a baseline for success and investigate potential flaws and weaknesses. We tried to gain insight into how faults in individual components propagated through the system and manifested as user-facing failures.

A. *Resilience Testing and Architectural Flaws*

The original TeaStore architecture (Figure 2) is composed of a set of interdependent microservices that rely on synchronous HTTP communication. The most significant architectural flaw, however, is the use of the monolithic Persistence service, which acts as a centralized data hub for nearly every other service in the system. This design choice creates a high degree of coupling, making it the system's most critical single point of failure. Our initial performance analysis revealed that under a sustainable load of 70 parallel users, the Persistence service was a significant bottleneck. The traffic graph (Figure 1) generated during this baseline test shows that a disproportionate 61.9% Teastore-webui and all requests from Teastore-auth were routed to this single service. This heavy concentration of traffic confirmed its critical role and its vulnerability to overload.

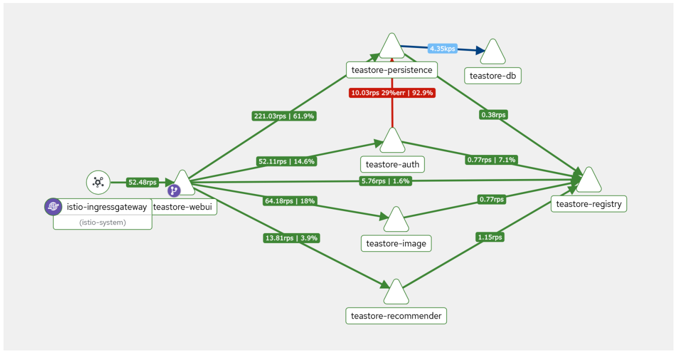

Fig. 1: Baseline Traffic Graph

Our testing method involved simulating the complete failure of one service at a time to observe the resulting violations of the 12 core functionalities defined in Table **??**.

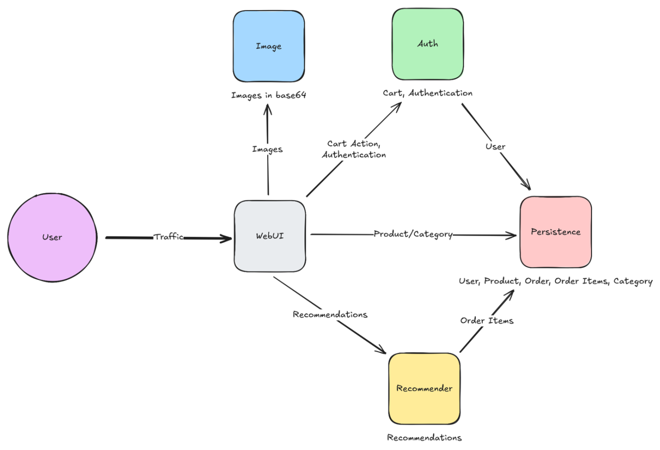

Fig. 2: Original Teastore Architecture

B. *Critical Failure Scenarios and Their Impact on Users*

Our experiments revealed a system with poor fault isolation, where some individual service failures can trigger cascading functionality outages.

1) The Monolithic Persistence Failure: The most critical scenario was the failure of the Persistence service. As the central data repository, its unavailability causes a cascading failure that violated 9 of the 12 core functionalities. Five Discovery features were unable to function because information regarding products and categories could not be fetched. All three Account features failed since user data and order data were not accessible. The Checkout process, naturally, was also impossible. Rather than degrading, the system effectively collapsed and rendered the application almost entirely useless.

2) The Authentication and WebUI Failures: A failure in the Auth service proved to be nearly as critical, violating 7 of the 12 functionalities. Although users could still browse products, any action that required a user session whether it was all the Shopping Cart features, the Account features or Checkout, was impossible. This showed that what was presumed to be a secondary service was critical to the core e-commerce workflow. Moreover, the WebUI serviced all traffic as a single point of entry. Its failure was total, violating all 12 functionalities.

3) Feature Failures: Failures in Image and Recommender services were less catastrophic but still resulted in a significantly degraded user experience. Each failure violated one core functionality (See product images and See recommendations, respectively). The system lacked any fallback mechanism, presenting users with broken image placeholders or empty recommendation sections.

Unavailable Service	Impacted Functionalities	Severity
WebUI	12/12	Application Crash
Persistence	9/12	Critical Failure
Auth	7/12	Critical Failure
Image	1/12	Feature Failure
Recommender	1/12	Feature Failure

TABLE II: Baseline Analysis Results

This analysis provided a clear baseline of the system's weaknesses. The tightly coupled design and the monolithic persistence layer ensured that faults were not contained, leading to a significant loss of functionality. This provided a strong motivation for the architectural refactoring detailed in the next section.

IV. Refactoring for User-Centric Resilience

After identifying the architectural issues and evaluating their impact on the user experience, we ranked them based on severity and addressed them in order of importance. Our implementation strategy focused on achieving the greatest resilience gains with the least complexity, especially for components critical to the user journey.

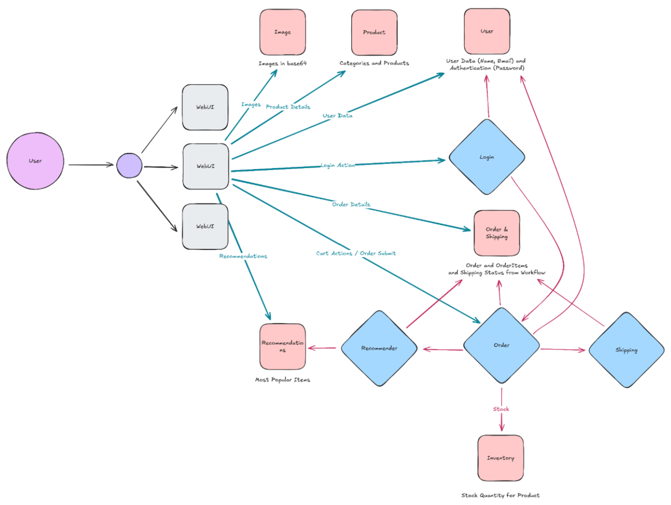

Fig. 3: Final Teastore Architecture

A. Replication

First, we distinguished between stateless and stateful components. Stateless services, such as the web frontend (WebUI), were straightforward to replicate horizontally. This replication alone provided a significant increase in fault tolerance with minimal implementation effort. If one instance of the WebUI goes down, traffic is automatically routed to a healthy instance, ensuring the application remains available to users.

B. Caching

Next, we conducted a usage analysis to understand access patterns and data dependencies. Since the TeaStore catalog is relatively small and access patterns showed repeated requests for the same items, implementing a cache layer was a highly effective step. This allowed frequently accessed product data to be served even when the persistence service was unavailable, maintaining a near-normal user experience in degraded scenarios.

C. Timeouts and Fallbacks

Another important measure was the careful configuration of timeouts. Properly set timeouts prevent long waits for the user and are essential for triggering fallback mechanisms in a timely manner.

Fallbacks vary depending on the functionality; for example, if the Recommender service is unavailable, the system falls back to showing random products to prevent an empty UI element or image service failures were handled by serving placeholders and deferring re-fetching to client-side JavaScript, effectively offloading recovery logic to the client and improving responsiveness. This offloading approach can be used in other scenarios too. These patterns provide a degraded but functional experience, which enhances fault tolerance.

While circuit breakers were also implemented, we found them to be of moderate importance. When timeouts are well configured, the added benefit of circuit breakers is primarily in accelerating failure detection and fallback initiation. Without the implementation of proper fallback mechanisms circuit breakers are a useless measure.

D. Multiple Databases

The original architecture's most critical vulnerability was the monolithic Persistence service, which created a single point of failure that could cause a system-wide collapse. A failure in this service led to the violation of 9 out of 12 core functionalities. Therefore the decomposition of the monolithic Persistence service was a more involved change.

This required significant effort, including data migration, schema redesign, and deployment of new database instances. The single Persistence service was broken down into smaller, domain-specific services, each with its own dedicated data store (Product, User, Order, Inventory, Recommender). However, once completed, this separation allowed us to build more targeted and efficient fallback strategies, making it a worthwhile investment, particularly in systems where a single database service cannot be reliably maintained.

By isolating the data stores, a failure in one service (e.g., the order database) no longer impacts unrelated functionalities like browsing products or viewing user profiles. This effectively limits the scope of impact from a failure, ensuring the problem remains contained.

Decoupling also improves resilience. With smaller, independent services, recovery is faster and more manageable. A single service can be restarted or scaled without affecting the rest of the system, allowing the application to return to a fully functional state more quickly.

For future implementations, using a managed cloud database service could simplify this process while maintaining high availability.

E. Health Probes

As part of the infrastructure improvements, we also introduced proper Kubernetes health probes—both liveness and readiness checks—for each service. These probes play a crucial role in maintaining overall system stability by allowing Kubernetes to automatically detect and replace failed containers or temporarily remove unhealthy services from the load balancer. In the original deployment, failing services often remained in a broken state, compounding outages and degrading user experience. By defining clear health criteria (such as successful Temporal connections or HTTP response checks), services can now fail fast and recover cleanly, improving fault isolation and reducing the blast radius of individual service failures.

F. Durable Execution

To address deeper structural limitations, we adopted Temporal as a durable execution platform for asynchronous logic. All key business processes—such as order processing, shipping, recommendations, and login—were refactored into Temporal Workflows. This architectural shift required a fundamental change in mindset, as operations like login and checkout no longer completed in a single synchronous request. Instead, they became long-running workflows that progress independently and automatically resume when dependencies become available.

However, the benefits of this change were substantial. Business logic is now resilient by default, thanks to Temporal's built-in state persistence, automatic retries, and guaranteed execution. For example, if the payment service is down when an order is submitted, the OrderWorkflow will simply pause and resume once the service recovers—without any data loss or manual intervention. Likewise, the LoginWorkflow will queue a request even if the user service is temporarily unavailable. From the user's perspective, the WebUI provides instant feedback ("processing...") and stays responsive, hiding backend service disruptions entirely.

This asynchronous, workflow-driven model proved particularly effective for decoupling frontend responsiveness from backend availability, and we highly recommend it for systems where business logic spans multiple services with potential failure points.

To fully leverage this model, we also encourage implementing idempotent transactions and the SAGA pattern in combination with Temporal. The original TeaStore architecture did not account for fault tolerance at the transaction level: if a mid-operation failure occurred—e.g., after a payment succeeded but before the order was persisted—the system could end up in an inconsistent state, or worse, silently drop the order. These issues are preventable.

By designing each step in a workflow to be idempotent, repeated executions (due to retries or resume events) do not cause duplicate or conflicting outcomes. This is essential in a distributed system, where retries and message replays are common.

In parallel, using the SAGA pattern within workflows ensures compensating actions are executed if a transaction cannot complete successfully. For instance, if an order is created but the shipping service remains unavailable beyond a timeout, the workflow can trigger a rollback—canceling the order, issuing a refund, or notifying the user appropriately.

Together, durable execution, idempotency, and SAGA orchestration form a powerful trio for building fault-tolerant distributed applications. They transform the system from one where partial failures cause user-facing errors or data loss, into one where recovery and consistency are handled automatically and transparently.

G. Client-side Caching

Finally, we introduced client-side caching using session cookies, storing lightweight user data such as profile information and recent orders. While this had a limited impact, since these features are less central to the primary user journey, it contributed to a smoother experience during temporary backend outages.

V. Evaluation and Discussion

The success of the refactoring effort was measured against the primary goal: transforming the TeaStore application from a tightly coupled system into one that could withstand failures and degrade gracefully from a user's perspective. This section evaluates the results of our implementation and discusses the broader implications of our user-centric approach.

A. Results

Our evaluation, based on simulating service failures, demonstrates a significant improvement in both fault tolerance and resilience. The prioritization of our refactoring efforts was directly informed by the impact of each failure on the 12 core user functionalities.

1) Replicated WebUI: The most critical and, therefore, highest-priority change was the replication of the stateless WebUI. This fault tolerance pattern addresses the single point of entry. While its failure still results in a total outage, replication ensures that traffic can be rerouted to a healthy instance, preventing a complete system collapse.

Unavailable Service	Before	After	Improvement
WebUI	12/12	0/12	100%
Workflows	5/12	1/12	80%
Image	1/12	0/12	100%
Recommender	1/12	0/12	100%
Product	3/12	0/12	100%
Order	2/12	0/12	100%
User	4/12	1/12	75%

TABLE III: Improved Architecture Results. Before and After measure the number of violated functionalities if the service becomes unavailable without and with the implemented fault tolerance measures.

2) Monolithic Persistence Service: Of equal importance was decoupling the monolithic Persistence service. This was a fundamental fault tolerance improvement. In the baseline architecture, a failure of the Persistence service made 9 of the 12 core functionalities unavailable (3/12 available). After refactoring into domain-specific data stores, a failure in a single Persistence service, such as the product database, only affects its direct dependents, leaving 11 of the 12 features available.

3) Durable Execution: The second-highest priority was the introduction of durable workflows with Temporal, which was a resilience enhancement. In the baseline, any failure during the checkout or login process would result in a complete failure of that operation. With Temporal, these stateful processes are guaranteed to complete. For example, a failure of the Auth service previously made 7 of the 12 functionalities unavailable (5/12 available). With the asynchronous login workflow, the system now maintains 10 of 12 functionalities, allowing users to continue their journey while authentication is retried in the background.

4) Caching and Timeouts: Medium-priority implementations included caching and retries/timeouts. Caching product, user, and order data provided a significant improvement in fault tolerance, allowing the system to serve useful data when backend services were down. This improved feature availability and user experience.

5) Fallbacks: Finally, the lowest-priority implementations were fallbacks for the Image and Recommender services. While these fault tolerance patterns have a smaller impact on the user journey, they significantly improve the user experience by preventing broken UI elements, increasing feature availability from 11/12 to 12/12 in their respective failure scenarios.

B. Discussion

The user-centric methodology for evaluating and improving resilience and fault tolerance proved to be an effective approach. It's primary advantage is that it provides a clear metrics that directly correlate to business value. We can concretely say that the failure of a specific service does or does not impact the user experience and establish, for example, that this failure would no longer prevent a user from completing a purchase. This provides a clear path for prioritization.

C. Recommendations

In summary, for general applications aiming to improve user-facing resilience, we recommend the following prioritized steps sorted by the impact-to-complexity ratio:

1) Replicate stateless components to eliminate single points of failure.
2) Introduce caching based on usage patterns, especially for frequently accessed and read-heavy data.
3) Configure timeouts carefully and pair them with appropriate fallback mechanisms.
4) Decouple critical stateful services, especially persistence layers, to enable independent scaling and fallback logic.
5) Adopt asynchronous execution platforms like Temporal to improve durability and fault recovery in business logic. Using orchestration over choreography in combination with this also increases the debugability of the system.
6) Implement Health and Readiness Probes to ensure quick restarts.
7) Use client-side recovery where possible, such as placeholders and deferred data loading.
8) (Optional) Use client-side caches selectively for non-critical data to improve responsiveness during partial outages.
9) (Optional) Apply circuit breakers if needed, but prioritize timeout configuration first.

These recommendations aim to improve resilience in a way that aligns with real user needs, focusing on maintaining core workflows, such as product browsing, cart interactions, and ordering, even in the face of partial failures.

This approach is applicable to other systems, such as user-facing applications like e-commerce platforms. The core principle of defining a user journey, identifying critical functionalities, and measuring their availability during chaos experiments can be universally applied.

VI. Conclusion

The goal of this project was to address a critical architectural flaw in the TeaStore application: the monolithic Persistence service that acted as a single point of failure that was causing catastrophic system-wide outages. Our analysis of the baseline architecture confirmed that a failure in this single service violated the majority of core user functionalities, rendering the application almost entirely unavailable.

We tackled this by conducting a large-scale refactoring based on a user-centered focus. The architectural transformation was twofold: first, we decoupled the monolithic Persistence service into multiple, domain-specific data stores, thereby improving fault tolerance by isolating failure domains. Second, we integrated the Temporal workflow engine to manage stateful business logic, which significantly enhanced resilience by ensuring that critical user-facing workflows like ordering and login would run to completion, even in the face of temporary failures. These changes were supported by the implementation of several resilience patterns, including caching, fallbacks, and replication.

The results confirm that our user-centric approach was effective. By focusing on the user experience, we were able to prioritize our efforts on the architectural flaws that caused the most significant disruptions. The refactored TeaStore is no longer a vulnerable application that collapses under failure but a system that degrades gracefully, maintaining critical functionality even during partial outages. Ultimately, this project successfully transformed the TeaStore into a fault-tolerant, and resilient cloud-native system, validating our approach as a strategy for building modern distributed applications.

References

[1] Thomas Anderson and Peter A Lee. “Fault tolerance terminology proposals”. In: *Reliable Computer Systems: Collected Papers of the Newcastle Reliability Project*. Springer, 1985, pp. 6–13.

[2] Nirjhor Anjum and Md Rubel Chowdhury. “Fault tolerance and load balancing algorithm in cloud computing: A survey.” In: *International Journal of Advanced Research in Computer and Communication Engineering* (2024). DOI: 10.2139/ssrn.4847308.

[3] Yao Cheng, Haitao Liao, and Elsayed A Elsayed. “From reliability to resilience: More than just taking one step further”. In: *IEEE Transactions on Reliability* 73.1 (2023), pp. 42–46.

[4] Ravi Jhawar, Vincenzo Piuri, and Marco Santambrogio. “A comprehensive conceptual system-level approach to fault tolerance in cloud computing”. In: *2012 IEEE International Systems Conference SysCon 2012*. IEEE. 2012, pp. 1–5.

[5] Muhammad Asim Shahid et al. “Towards Resilient Method: An exhaustive survey of fault tolerance methods in the cloud computing environment”. In: *Computer Science Review* 40 (May 2021), p. 100398. DOI: 10.1016/j.cosrev.2021.100398.