NextGen Scientific Publication
American International Journal of Computer Science and Technology
Volume 4 Issue 2, Pg. No. 49-59, AIJCST-V4I2P106, 2022
https://doi.org/10.63282/3117-5481/AIJCST-V4I2P106

Original Article

In-Memory Token Caching and Distributed Schema Validation Pipelines for High-Velocity API Intermediaries

***Chandra Kiran Yelagam**
Independent Researcher, USA.

Abstract:

Pharmacy benefit intermediaries must validate high-velocity eligibility tokens and membership payloads at low latency without loosening structural or semantic checks. This paper proposes a distributed validation architecture built from in-memory token caching, asynchronous JSON schema validation, regular-expression boundary checks and high-throughput payload deserialization. Caches on the intermediary nodes avoid repeated token verification. An asynchronous pipeline checks each membership payload against predefined schemas and semantic constraints, and rejects malformed fields, invalid token formats, missing attributes and inconsistent eligibility data before the request reaches downstream patient intake. Regular-expression checks on token patterns, field lengths and input types add a second, lexical layer of defense. Because the components are modular, processing load can be spread across nodes and a failure in one stage stays contained. The design also specifies how cache consistency, validation accuracy and fault isolation are maintained, so that the intermediary can interoperate with other healthcare systems. A cache hit is never treated as proof of authorization. The architecture has not been benchmarked; we propose evaluating it on latency, throughput, and cache hit rate, validation correctness and error-rejection rate under varying workloads.

Keywords:

Pharmacy Benefit Intermediaries, API Validation, In-Memory Token Caching, JSON Schema Validation, Asynchronous Processing, Payload Deserialization, Healthcare Interoperability.

Article History:

Received: 12.02.2022

Revised: 27.02.2022

Accepted: 05.03.2022

Published: 21.03.2022

1. Introduction

Providers, pharmacy benefit managers (PBMs), insurers and other intermediaries exchange eligibility, membership and administrative data through distributed application programming interfaces (APIs). These interfaces sit in the path of patient intake, insurance eligibility checks, benefit determination and claims preparation. As transaction volume and complexity grow, the API middleware has to answer quickly and still guarantee that payloads are structurally and semantically consistent and that authorization data is obtained reliably. A verification lapse at this layer propagates. Downstream services see duplicate requests and processing faults, and patients see the disruption. Caching, distributed processing, payload validation and resource use therefore have to be designed together rather than tuned one at a time.

In-memory caching is a natural way to reduce recomputation and keep high-velocity APIs responsive. Holding frequently used information briefly in memory avoids repeated database access, remote service calls and authentication operations, each of which is expensive when done per request. Li et al. [1] proposed adaptive priority-based cache replacement with prediction-based prefetching for edge computing, and showed that cache management informed by access patterns and resource limits can improve processing speed. Their setting is edge computing rather than healthcare eligibility APIs, but the same cache-management view is a reasonable

Copyright by the Author(s). This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-NC-ND 4.0) License (https://creativecommons.org/licenses/by-sa/4.0/)

starting point for low-latency intermediaries. In pharmacy benefit processing it can be applied to token data and validation metadata, provided cached entries are subject to expiration, invalidation and security rules.

Caching also involves a trade-off among response time, storage space and retention time. Carra et al. [2] studied elastic provisioning of cloud caches using a cost-aware time-to-live (TTL) scheme, which makes the case for driving cache allocation and retention decisions by operational considerations. This matters for eligibility tokens, whose validity depends on time and whose membership data can change. An intermediary cannot assume that an entry in memory is still valid. Cache lifetime should instead follow token lifetime, revocation rules and the freshness the underlying healthcare transaction requires. The objective is to cut unnecessary processing without compromising authorization or data consistency; a high hit rate is secondary.

Cache hierarchy also shapes how a distributed service behaves. Trinh et al. [3] proposed C3C, a static content-based three-level web cache that organizes caching in layers. A layered structure suits systems whose processing stages have different access patterns and performance needs. In a healthcare API, token lookup, schema lookup and payload validation can be treated as separate stages. Reusable token metadata can sit in an in-memory cache, while incoming membership payloads are checked structurally and semantically before being routed to downstream services. Healthcare payloads cannot be handled like static web content, though. They must be checked against transaction context, schema version, token status for the same credential and changing membership information. The architecture therefore has to separate reusable metadata from transaction-specific data, so that caching never bypasses required validation.

Distributed caching also depends on coordination among service components across the network. Fang et al. [4] described a collaboration scheme in which an edge internet service provider works with content providers to improve content delivery, and their results indicate that cache placement and co-location affect delivery efficiency in a distributed setting. For API intermediaries, the analogous case is many service instances behind load balancers, where a request may reach any node and yet token management must stay coherent and validation results predictable. A distributed design should limit inter-service communication, scale request processing, and absorb cache misses or node crashes without interrupting required validation.

Existing caching studies remain relevant, but a pharmacy benefit intermediary needs more than a cache replacement policy or a content-delivery strategy. Incoming membership payloads and eligibility tokens must be both structurally and semantically correct. JSON schema validation can detect missing fields, wrong data types and deviations from predefined payload formats, and boundary checks with format constraints can catch malformed values. Token verification, for its part, must establish independently that a credential is correct and within scope. Combining these steps in a distributed pipeline poses a design problem: validation has to be complete without duplicating effort or adding excessive latency.

This paper proposes an architecture that couples in-memory token caching with asynchronous JSON-schema-based payload validation and boundary-level checks for high-velocity healthcare APIs. Cached token metadata removes repeated verification work, while every payload still passes the full validation sequence, so that eligibility and membership data reaching downstream services remain accurate and consistent. The paper does not report measurements. It defines the components, their interactions and their failure behavior, and names the measures against which an implementation should be tested, namely response latency, throughput, cache performance, error detection and validation overhead under varying request loads.

2. Related Work

Li et al. [1] pair adaptive priority-based cache replacement with prediction-based prefetching for edge computing, so that fewer objects are retrieved twice. The same principles can guide how frequently used eligibility-token data is stored in an API intermediary, but only up to a point. A token entry also expires, can be revoked and must still authorize correctly, none of which a content cache has to consider.

Carra et al. [2] tie cache allocation and retention period to operating cost. In a distributed pharmacy benefit system, keeping frequently used token metadata reduces verification overhead, but keeping too much of it, or keeping it too long, leaves stale state. Retention therefore has to be balanced against token lifetime, freshness of membership data and invalidation. Unlike a general cost-aware cache, an eligibility intermediary must also guarantee that performance tuning never bypasses an authorization check.

The three-level organization of C3C [3] shows how caches at different levels can support efficient retrieval, and it suggests that an API intermediary could likewise separate token information, reusable schema definitions and individual payloads. The split makes the architecture easier to reason about. A healthcare payload, however, still needs contextual validation and cannot be judged from cached data alone, so the framework proposed here treats caching and validation as separate responsibilities that have to be coordinated.

Fang et al. [4] show that cache location and cooperation between providers affect distributed service delivery. Healthcare APIs deployed across many intermediary instances face the same issue, since request distribution and inter-service communication affect performance. They also carry an extra requirement, because every node must enforce the same security policies and produce the same validation results. Cache coordination therefore has to be designed together with failure management and validation controls.

Chao [5] developed a web cache replacement strategy that combines Greedy-Dual-Size-Frequency (GDSF) with support vector machine (SVM) prediction of re-access probability, so that replacement follows estimated future demand. For eligibility-token caching, access frequency helps identify good candidates for retention but should not be the only retention rule. Predicted reuse should rank below token expiration, revocation status and security requirements.

Baskaran and Kalaiarasan [6] combined clustering-based web prefetching with caching, using SVM learning. Their study shows how access-pattern analysis can support anticipatory retrieval. Prefetching can help conventional web systems, but in healthcare intermediaries it must be tightly constrained, since speculative access to sensitive or transaction-related data adds load and security risk. The challenge is to reuse metadata efficiently without skipping request-time validation.

Olanrewaju and Azman [7] improved web proxy cache performance using wrapper-based feature selection with Naive Bayes and J48 classifiers. Their results show that informative features can be selected for learning-based cache policies. In a healthcare API environment, relevant features could include access frequency, token lifetime and cache-entry size. Such features could inform cache-management decisions, but schema validity and authorization decisions must still rest on explicit rules, not statistical prediction.

Abdalla et al. [8] applied supervised machine learning with feature selection to predict web objects in proxy caching, using request-related characteristics to predict object reuse. For pharmacy benefit intermediaries, this suggests that frequently reused, non-sensitive metadata could be cached more effectively. Prediction alone, however, cannot decide whether an incoming membership payload is structurally and semantically correct and consistent with the current transaction.

Ibrahim et al. [9] used J48 decision trees and Naive Bayes to build intelligent cooperative web caching policies for media objects in structured peer-to-peer systems. Their work addresses both cooperation among distributed caching components and the use of supervised learning for caching decisions. These ideas could carry over to multi-instance API architectures, where a coordinated caching policy spans distributed nodes. In healthcare deployments, though, cooperation must not leak sensitive information or expose inconsistent token states.

Benadit et al. [10] used a tree-augmented Naive Bayes classifier to improve proxy cache performance, which adds to the broader line of work on probabilistic classification for cache management. Pernabas et al. [11] studied how a weighted random-indexing-based data-mining classifier can improve greedy web proxy caching. Both papers show that richer decision models can improve caching decisions. Because they target cache performance, however, they leave out two things a healthcare API pipeline also needs: deterministic payload verification and secure token verification.

Ma et al. [12] developed a weighted GDSF cache replacement algorithm that takes several factors into account when choosing what to replace. In related work, Ma et al. [13] proposed an improved web cache replacement algorithm based on weighting and cost. Both works argue that replacement decisions should weigh several operating factors instead of access frequency alone. That idea carries over to token caches, where security requirements and time constraints should be able to override other ranking criteria in replacement.

Petridou et al. [14] investigated time-aware clustering of web users and found that time attributes help reveal access behavior. Huang and Hsu [15] mined web logs to improve the hit ratios of prefetching and caching. Together these studies show that temporal

patterns and request history can be used to manage resources more efficiently. Analysis of access patterns and better cache-hit ratios still say nothing directly about the structural and semantic correctness of healthcare membership payloads.

Prior work thus centers on cache replacement, predictive access, resource provisioning and distributed cache coordination. We found no architectural specification that combines these with asynchronous schema validation, boundary verification and secure eligibility-token management in a healthcare API pipeline. The framework in Section 3 addresses that combination, and it uses in-memory caching only where mandatory verification is still enforced.

3. Distributed Low-Latency Validation Architecture for Healthcare Eligibility APIs

3.1. Framework Overview and Design Principles

The framework combines distributed, asynchronous in-memory token caching, JSON schema validation, regular-expression boundary validation and lightweight payload deserialization to support high-velocity API processing in pharmacy benefit intermediary systems. Its main goals are to cut unnecessary processing latency and to preserve the structural and semantic integrity of incoming eligibility tokens and membership payloads. The architecture is modular and horizontally scalable, with caching and validation as independent but coordinated processing units.

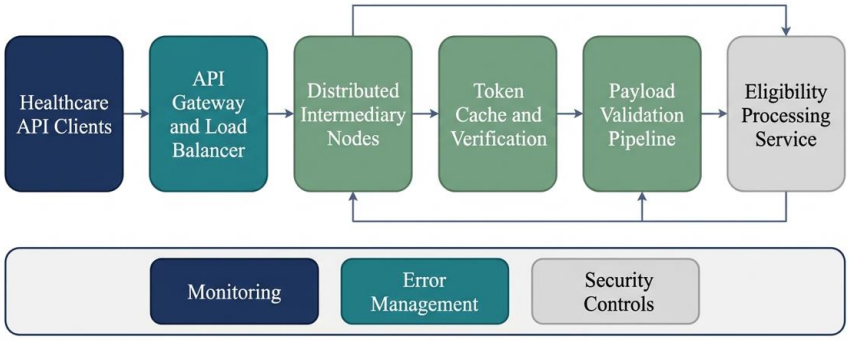


Figure 1. Overall Architecture of the Distributed Healthcare API Validation Framework

Figure 1 shows the overall architecture and how caching, validation and downstream processing components interact. It has six main segments: API request ingestion, token verification and distributed caching, payload deserialization, asynchronous structural validation, semantic and boundary validation, and controlled response management. Requests first reach an API gateway or load balancer, which distributes them across available intermediary nodes. Each node runs the required validation steps and contacts shared or distributed cache infrastructure according to the configured consistency policy. Requests that pass all checks go on to downstream patient intake or eligibility-processing services, and invalid requests are rejected with controlled error responses.

The design follows four principles: low-latency processing, validation completeness, distributed scalability and secure failure management. Caching reduces redundant token checks, asynchronous processing lowers resource consumption, and modular validation isolates individual processing tasks. A cache hit never bypasses authorization checks and is never taken as proof that a token is legitimate.

3.2. Distributed Token Caching and Consistency Management

The token-caching component keeps metadata for frequently used tokens in memory, reducing database queries and remote verification calls. When an API request arrives, the intermediary extracts the token, runs preliminary format checks and then consults the cache. The cache key is a cryptographic digest of the token, not the credential itself, so that sensitive authentication data is less likely to leak.

A cache lookup ends in one of three ways: a valid and sufficiently fresh entry, a miss, or an expired or invalid entry. A valid entry supplies reusable verification metadata, subject to the authorization policy. A miss triggers a check with the authoritative identity or authorization service. Expired entries are deleted or rebuilt, and invalid credentials are rejected. Configurable time-to-live (TTL) values, capacity limits and eviction policies let operators balance memory consumption against lookup effectiveness.

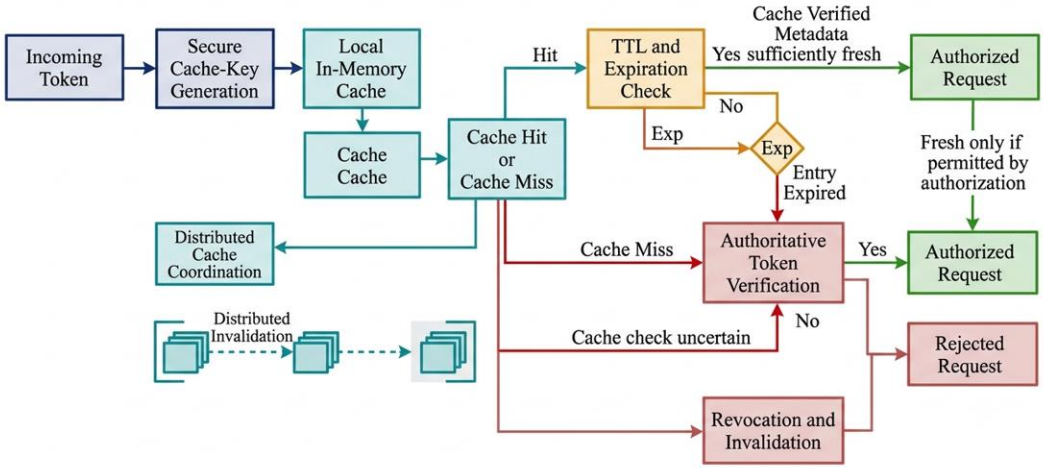


Figure 2. Distributed Token Caching and Consistency Management

Figure 2 depicts the distributed token-caching mechanism, including cache lookup, authoritative verification, expiration handling and cross-node consistency management. A distributed deployment may keep local in-memory caches on individual intermediary nodes, together with a shared mechanism for invalidation events. Local caches give the fastest access, and distributed invalidation keeps nodes from acting on conflicting token state. Cache entries should carry version identifiers, relevant timestamps, expiration information and issuer or scope metadata where appropriate. High-risk credentials may need shorter retention periods or online revocation checks.

The cache-management logic draws on the adaptive replacement of Li et al. [1] and the cost-aware TTL management of Carra et al. [2]. Security constraints, however, take priority over cache-hit optimization. When authoritative verification is unavailable and the credential cannot be fully trusted, the framework applies a predefined fail-closed policy and does not accept uncertain credentials.

3.3. High-Throughput Deserialization and Asynchronous Processing

Once the token has been checked, the request-processing component converts the incoming JSON payload into a structured object that can be validated. Before committing significant processing resources, the deserialization layer checks the payload encoding, permitted content type, maximum payload size, nesting depth and allowed data formats. These checks keep malformed or oversized payloads from consuming disproportionate memory or CPU.

Asynchronous execution decouples request reception from the computationally heavy validation work. Requests enter a bounded processing queue, so the intermediary can handle multiple workloads concurrently without letting the backlog grow without limit. Independent validation operations can run in parallel when they have no dependencies on one another, and the final response waits until all required checks have completed.

Backpressure applies when queue utilization exceeds set limits. When capacity runs short, the system may delay requests, rate-limit them or return a temporary service-unavailable message. Timeouts and cancellation keep stalled operations from holding resources indefinitely, and connection pooling and object reuse reduce repeated allocation and communication overhead.

The pipeline also attaches request identifiers and structured diagnostic metadata so that processing can be traced across distributed nodes. Routine logs leave out sensitive token values and membership attributes. Bounded parallelism, careful deserialization and asynchronous task execution are intended to keep processing time predictable as transaction load varies. Figure 3 outlines the asynchronous payload-processing pipeline and shows how bounded queues and concurrent validation tasks support controlled request handling.

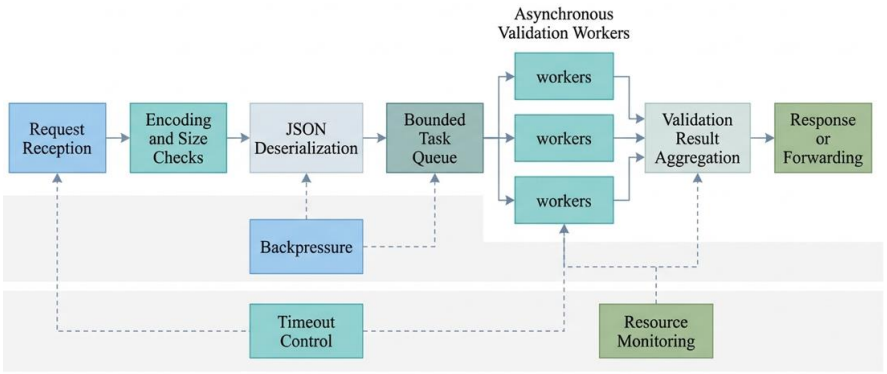


Figure 3. Asynchronous Payload Processing and Validation Pipeline

3.4. Layered Structural, Semantic, and Boundary Validation

The validation subsystem applies several complementary checks to every incoming membership payload. The first stage is structural validation, which includes JSON syntax verification and schema-based checking. A JSON schema defines mandatory fields, permitted data types, field formats, lists and any length or range restrictions. Schema versions are stored explicitly so that each payload is validated against the definition its contract names.

Schema definitions can be kept in a cache, as token metadata is. Schema-cache entries carry a version key and expiration or invalidation policies. When a schema is updated, this prevents some nodes from validating requests against an outdated definition. If compatibility between two schema contracts is uncertain, the schema compatibility mechanism either selects the correct authoritative schema or rejects the request. Figure 4 presents the layered validation workflow and shows how structural, semantic and boundary checks together determine whether a payload is acceptable.

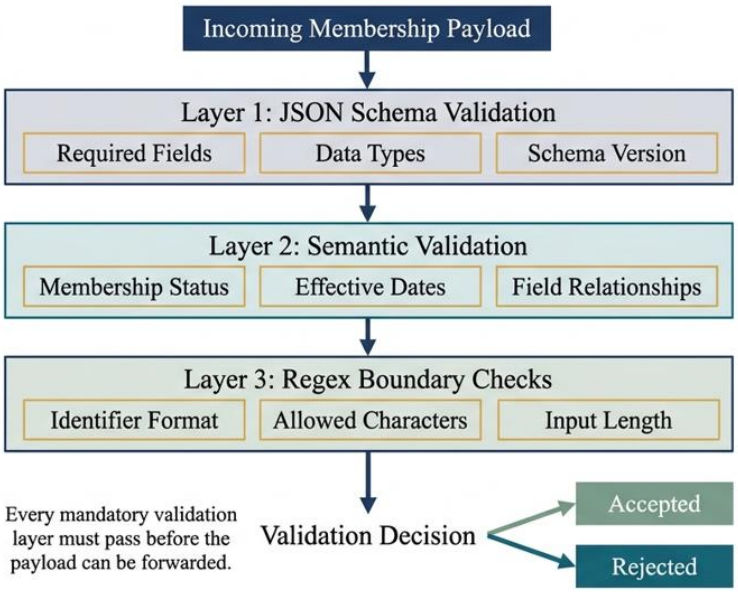


Figure 4. Layered Structural, Semantic, and Boundary Validation

Semantic validation is the second stage. It checks field relationships and business constraints that structural rules cannot express well. Examples include consistency between membership status and effective dates, validity of the relationship between subscriber and dependent identifiers, and agreement between eligibility periods and transaction conditions. When external verification is needed, the stage queries the appropriate authoritative service and does not assume that a well-formed payload is valid.

The third stage applies regular-expression boundary testing to fields that need specific lexical rules. These checks can verify permitted identifier characters, date-string formats and other input restrictions set by the contract. Expressions should be anchored to the whole input and kept simple, with complexity limits, to avoid excessive backtracking. Regex validation complements JSON schema validation and semantic business rules and does not replace them.

Validation results are summarized in a structured decision record containing the request identifier, the validation stage, the decision and the applicable error category. A request is forwarded only after all required checks have passed. This layered approach lowers the chance that malformed, inconsistent or unnecessary data reaches healthcare services.

3.5. Error Isolation, Security, and Controlled Response Management

Response management separates authentication errors, malformed JSON, schema errors, semantic errors, boundary errors, rate-limit errors and temporary infrastructure errors. Each category has its own response code and maps to a uniform error structure. Responses give legitimate API consumers enough information to correct their requests, without exposing internal implementation details, credentials or sensitive membership information.

The architecture isolates failures of cache nodes, validation workers and downstream services from one another. A cache-node failure can trigger a supervised fallback to authoritative verification, subject to timeout and availability rules. A validation-worker failure must never result in an unvalidated request being forwarded. Request deduplication where applicable, circuit breakers, limited retries and health monitoring help contain cascading failures. Figure 5 shows the fault-isolation and error-handling architecture, which contains processing failures while preserving authorization and validation requirements.

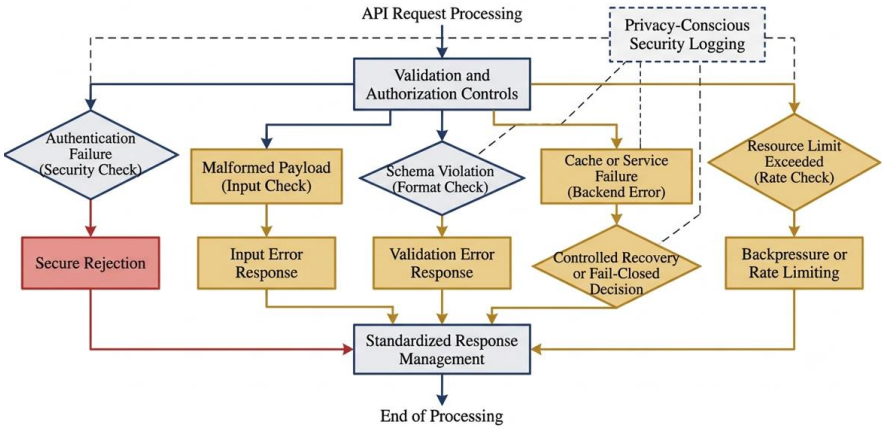


Figure 5. Fault Isolation and Security-Governed Error Handling

Security controls include encrypted transport, restricted access to caches, secure configuration, least-privilege service identities and careful handling of sensitive data. Observability metrics cover request latency, throughput, cache-hit latency, schema-validation latency, rejection rate, queue length and infrastructure error rate. These measurements support performance analysis and bottleneck identification without compromising validation goals.

3.6. Operational Workflow and Scalability

The workflow starts with acceptance of the API request and basic input validation, followed by a token-cache lookup and, where necessary, authoritative validation. The payload is then deserialized and passed to asynchronous structural, semantic and boundary validation. Requests that satisfy all required checks go to downstream services such as eligibility or patient intake, and invalid requests are discarded in a controlled way. Operational metrics and security events are recorded throughout.

Horizontal scaling works by adding intermediary nodes behind a load balancer and adjusting worker capacity to the observed workload. Cache capacity, TTL settings, concurrency limits and schema-validation policies can each be configured independently. The framework could then be compared with other setups on average and tail latency, requests served per second, cache hits, validation correctness, malformed-payload rejection and resource utilization.

The design rule running through this section is that caching and concurrency may shorten the path through the intermediary but may never remove a validation step.

4. Architectural Validation and Functional Adequacy Assessment

4.1. Structural Coherence and Functional Coverage

The framework is evaluated through architectural analysis, functional coverage and fit with the operational requirements of distributed pharmacy benefit intermediaries. The analysis considers whether token caching, asynchronous deserialization, structural validation, semantic validation and controlled response management form a consistent and logical way to handle healthcare eligibility requests. Rather than treating each component as a separate optimization, it looks at the dependencies between components and how they affect one another in producing reliable API processing.

The review shows that authorization management and payload validation are clearly separated. Token caching supports reuse of valid verification metadata, while semantic verification and JSON validation independently decide whether the membership data is acceptable. This separation keeps cache availability from being treated as evidence that a payload is correct. Regular-expression checks likewise complement structural validation: they impose field-specific input constraints but do not replace business-level validation.

The framework also covers normal processing and several failure conditions: invalid input, cache misses, expired credentials, schema incompatibility and temporary service failures. Each condition has a defined processing response, which removes ambiguity about how components interact when something goes wrong. Explicit schema versions and controlled cache invalidation further help keep distributed intermediary nodes consistent.

4.2. Comparative Assessment of Architectural Trade-offs

The framework is compared qualitatively with three conventional architectures: direct token verification with synchronous validation, cache-based API processing with limited payload verification, and asynchronous validation without coordinated caching.

Direct verification with synchronous validation has a simple control flow but depends heavily on the authoritative verification service. Sequential handling and repeated remote verification can increase communication overhead and limit concurrency. Cache-based processing reduces duplicate verification, but partial payload validation lets semantically inconsistent yet structurally valid membership data flow downstream. Asynchronous validation without coordinated caching gives more flexibility in processing but may still carry unnecessary verification overhead when token checks are poorly managed.

The proposed framework keeps the full validation of the first design, the reduced verification traffic of the second and the processing flexibility of the third. Distributed caching reduces redundant verification, asynchronous processing runs inside bounded queues, and layered validation checks structure, semantics and boundaries independently of how requests are scheduled. The price is complexity. Cache coordination requires invalidation mechanisms, asynchronous execution complicates scheduling, and each additional validation stage raises implementation and maintenance cost. The framework contains these costs through explicit validation dependencies, bounded queues, modular components and a security-driven cache policy.

4.3. Security Assurance and Operational Applicability

The final check is whether the architecture can withstand the conditions that stress distributed healthcare API processing. It handles uncertain authorization claims conservatively, restricts the processing of expired credentials and forbids forwarding unauthenticated payloads, which sets clear security limits. Regular-expression checks and restricted deserialization also reduce the risk of resource exhaustion and malformed input.

Making these optimizations work requires consistent configuration across intermediary nodes, an appropriate cache-consistency policy, handling of schema changes, and traceability that does not expose sensitive healthcare information. These requirements add administrative overhead but are needed for dependable deployment.

The analysis establishes conceptual and functional adequacy only. It shows that low-latency token management and full payload validation can be combined in one architecture without relaxing mandatory security or data-integrity constraints. Claims of actual performance gains and production behavior would need independent empirical validation under realistic operating conditions.

5. Architectural Implications and Design Trade-offs in Healthcare API Intermediation

In a pharmacy benefit intermediary, performance work has to sit alongside validation controls that it cannot weaken. The design places in-memory token caching, asynchronous request processing and layered payload validation in one model, and treats processing efficiency, authorization correctness, membership-data integrity and operational resilience as separate objectives rather than as by-products of low latency.

One implication is that token verification is kept separate from payload verification. Caching can remove redundant verification, but any cached metadata remains subject to expiration, revocation and consistency rules. This matters in healthcare, where outdated authorization data can lead to wrong access decisions. The architecture therefore favors security-governed cache reuse over maximizing cache hits.

Using JSON schema validation, semantic validation and regular-expression boundary checks together also avoids reliance on any single validation mechanism. Conforming to the structure does not show that business rules are met, and a matching pattern does not show that eligibility relationships are correct. Because each layer verifies different information, together they cover more cases and reduce the chance that inconsistent data reaches downstream patient intake systems.

The architecture does introduce trade-offs in cache synchronization, schema-version control, coordination of asynchronous tasks and operational complexity. Distributed invalidation can add communication overhead, and additional validation stages must be managed through dependency control. Implementation policies should therefore be tuned to transaction sensitivity, authorization needs and infrastructure capability, not to processing efficiency alone.

The framework also bears on healthcare interoperability. Explicit validation contracts, standardized error responses and controlled failure management can improve consistency among heterogeneous services. Privacy-conscious logging and careful resource management also support operational governance without exposing sensitive membership information.

The main limitation is that the framework presents architectural principles, not proven production results. Actual performance will depend on implementation quality, workload characteristics, cache-consistency guarantees and the connection to authoritative eligibility services. Future research should examine these factors through deployment-like validation and comparative empirical studies.

6. Evolution toward Intelligent and Adaptive Healthcare API Intermediation

6.1. Adaptive Cache Intelligence and Consistency Governance

Future extensions could make cache management adaptive, adjusting token-metadata retention dynamically according to access patterns, sensitivity of the authorization data, revocation needs and transaction type. Rather than relying on time-to-live policies alone, context-aware cache management could separate widely accessed, low-risk metadata from security-sensitive entries that need stricter verification. Event-based coordination and version-aware cache coordination could strengthen distributed invalidation. Any such extension must keep authoritative authorization controls in place, and predictive cache decisions must never override credential validity or security policy.

6.2. Context-Aware Validation and Schema Evolution

The validation pipeline could be extended with context-sensitive policy engines that understand membership relationships, transaction-specific constraints and changing healthcare data contracts. Automated schema versioning could make transitions between API versions controlled and handle deferred fields and deprecated attributes. Future implementations could also adopt standard healthcare interoperability formats, such as FHIR-based resources, where applicable. More advanced validation policies could combine declarative business rules with JSON schema rules to improve maintainability and consistency across distributed services. Any automation of this kind should come with explicit rules, traceability and approval of validation policy.

6.3. Resilient Processing and Privacy-Preserving Observability

Resilience work could add adaptive backpressure, workload-sensitive request staging and better fault isolation across cache and validation workers. Distributed tracing could give end-to-end visibility into processing dependencies, leaving sensitive tokens and unnecessary patient identifiers out of diagnostic records. Privacy-preserving observability could support operational analysis of

aggregated events and telemetry under suitable security controls. Configuration auditing could also be automated to detect configuration differences between intermediary nodes in schema versions and verification rules.

6.4. Deployment-Oriented Validation and Interoperability

A further direction is to evaluate the framework in representative pharmacy benefit workflows and heterogeneous deployment environments. Future studies could examine security behavior under token revocation, schema change, node crashes and floods of simultaneous requests. Comparison with traditional architectures would help establish the feasibility and limits of the design. Work could also extend to integration with standardized API security mechanisms and healthcare interoperability interfaces to assess deployment readiness.

None of these extensions should relax the rules on which the current design rests. Authorization correctness, payload integrity and privacy protection come before adaptivity.

7. Conclusion

This paper presented a distributed architecture for low-latency API validation in pharmacy benefit intermediaries in which token verification is kept apart from payload verification. Cached token metadata cuts repeated verification, but its reuse is governed by expiration, invalidation and consistency rules, and when authoritative verification is unavailable the intermediary fails closed. Every payload passes structural, semantic and boundary validation before downstream processing, whether or not its token was found in the cache. Bounded queues and asynchronous deserialization keep overlapping requests from exhausting resources.

The qualitative comparison in Section 4 identified the costs of this design in cache coordination, schema adaptation, asynchronous scheduling and failure management. The simpler alternatives avoid those costs only by repeating remote verification on every request or by letting structurally valid but semantically inconsistent data through.

The architecture has not been validated in production or through quantitative performance measurement. Its actual benefits will depend on implementation choices, workload characteristics, cache-consistency options and the connection to authoritative eligibility services. Future work should measure its behavior under representative transaction and authorization loads, during schema evolution and under distributed infrastructure failures.

The same separation of cached authorization state from mandatory payload validation may also apply to other healthcare integration settings that require high throughput.

References

[1] Li, C., Song, M., Du, S., Wang, X., Zhang, M., & Luo, Y. (2020). Adaptive priority-based cache replacement and prediction-based cache prefetching in edge computing environment. Journal of Network and Computer Applications, 165, Article 102715. https://doi.org/10.1016/j.jnca.2020.102715

[2] Carra, D., Neglia, G., & Michiardi, P. (2020). Elastic provisioning of cloud caches: A cost-aware TTL approach. IEEE/ACM Transactions on Networking, 28(3), 1283–1296. https://doi.org/10.1109/TNET.2020.2980105

[3] Trinh, T., Wu, D., & Huang, J. Z. (2019). C3C: A new static content-based three-level web cache. IEEE Access, 7, 11796–11808. https://doi.org/10.1109/ACCESS.2019.2892761

[4] Fang, C., Yao, H., Wang, Z., Si, P., Chen, Y., Wang, X., & Yu, F. R. (2019). Edge cache-based ISP-CP collaboration scheme for content delivery services. IEEE Access, 7, 5277–5284. https://doi.org/10.1109/ACCESS.2018.2889787

[5] Chao, W. (2020). Web cache intelligent replacement strategy combined with GDSF and SVM network re-accessed probability prediction. Journal of Ambient Intelligence and Humanized Computing, 11(2), 581–587. https://doi.org/10.1007/s12652-018-1109-4

[6] Baskaran, K. R., & Kalaiarasan, C. (2016). Improved performance by combining web pre-fetching using clustering with web caching based on SVM learning method. International Journal of Computers Communications & Control, 11(2), 167–178.

[7] Olanrewaju, R. F., & Azman, A. W. (2017). Enhancement web proxy cache performance using wrapper feature selection methods with NB and J48. IOP Conference Series: Materials Science and Engineering, 260(1), Article 012012. https://doi.org/10.1088/1757-899X/260/1/012012

[8] Abdalla, A., Sulaiman, S., & Ali, W. (2015). Intelligent web objects prediction approach in web proxy cache using supervised machine learning and feature selection. International Journal of Advances in Soft Computing and Its Applications, 7(3), 1–19.

[9] Ibrahim, H., Yasin, W., Udzir, N. I., Hamid, A., & Wati, N. A. (2016). Intelligent cooperative web caching policies for media objects based on J48 decision tree and Naïve Bayes supervised machine learning algorithms in structured peer-to-peer systems. Journal of Information and Communication Technology, 15(2), 85–116.

[10] Benadit, P. J., Francis, F. S., & Muruganantham, U. (2015). Improving the performance of a proxy cache using tree augmented Naive Bayes classifier. Procedia Computer Science, 46, 184–193. https://doi.org/10.1016/j.procs.2015.02.010

[11] Pernabas, J. B., Fidele, S. F., & Vaithinathan, K. K. (2019). Enhancing greedy web proxy caching using weighted random indexing based data mining classifier. Egyptian Informatics Journal, 20(2), 117–130. https://doi.org/10.1016/j.eij.2019.01.001

[12] Ma, T., Qu, J., Shen, W., Tian, Y., Al-Dhelaan, A., & Al-Rodhaan, M. (2018). Weighted greedy dual size frequency based caching replacement algorithm. IEEE Access, 6, 7214–7223. https://doi.org/10.1109/ACCESS.2018.2790381

[13] Ma, T., Hao, Y., Shen, W., Tian, Y., & Al-Rodhaan, M. (2018). An improved web cache replacement algorithm based on weighting and cost. IEEE Access, 6, 27010–27017. https://doi.org/10.1109/ACCESS.2018.2829142

[14] Petridou, S. G., Koutsonikola, V. A., Vakali, A. I., & Papadimitriou, G. I. (2008). Time-aware web users' clustering. IEEE Transactions on Knowledge and Data Engineering, 20(5), 653–667. https://doi.org/10.1109/TKDE.2007.190741

[15] Huang, Y.-F., & Hsu, J.-M. (2008). Mining web logs to improve hit ratios of prefetching and caching. Knowledge-Based Systems, 21(1), 62–69. https://doi.org/10.1016/j.knosys.2006.11.004