NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 1 Issue 5, Pg. No. 10-16, AIJCST-V1I5P102, 2019
https://doi.org/10.63282/3117-5481/AIJCST-V1I5P102

Original Article

A Standardized Microservices Framework for Electronic Toll Collection Back-office Systems: Reference Architecture, Governance, and Migration Strategy

***Sarath Babu Gosipathala**
Independent Researcher, USA.

Abstract:

Electronic toll collection agencies operate back-office systems that process trip transactions, manage violations and pay-by-mail correspondence, maintain customer accounts, post finance records, track inventory and tag assets, and generate regulatory reports. Agencies such as OCTA/RCTC, NTTA, and CTRMA-style authorities commission these systems independently, and each engagement typically produces a monolithic ASP.NET MVC, .NET Core, and MS-SQL Server platform built largely from scratch. Repeated custom development across agencies with materially similar functional requirements creates redundant engineering effort, extends the interval between a request for proposal and system deployment, and complicates long-term maintainability. A standardized microservices framework addresses this pattern by decomposing common back-office functions into independently deployable services governed by well-defined RESTful contracts, a shared data model, a centralized identity and access layer, and unified logging and monitoring. Agency-specific business rules attach to the shared core through configurable modules rather than through system-wide rewrites, allowing each agency to retain its distinct policy, fee, and violation-handling requirements. Containerization and cloud-native deployment practices support consistent packaging, scaling, and release management across agency environments. Applying this framework across projects that would otherwise be rebuilt independently shortens the interval from contract award to operational deployment, reduces gap-analysis effort during each engagement, and improves the consistency of updates applied across the agency portfolio. The practical significance extends to procurement teams, systems integrators, and public agencies seeking predictable delivery timelines, lower lifecycle cost, and a governed path for migrating legacy monolithic platforms toward a common, extensible architecture.

Keywords:

Micro Services Architecture, Toll Collection Systems, Back-office Modernization, Reference Architecture, Cloud-native Governance.

Article History:

Received: 23.07.2019

Revised: 08.08.2019

Accepted: 20.08.2019

Published: 06.09.2019

1. Introduction

Electronic toll collection has shifted over the past decade from lane-based cash handling toward fully account-based, all-electronic tolling supported by extensive software back-office infrastructure. The back-office system (BOS) of a modern tolling agency is responsible for converting raw transponder and license-plate-image trip records into billable transactions, applying agency-specific toll and fee schedules, managing customer accounts and prepaid balances, issuing violations and pay-by-mail correspondence for

unregistered trips, posting financial entries to agency ledgers, tracking transponder and tag inventory, and producing operational and regulatory reports. Delivering these capabilities has historically meant assembling a bespoke platform for each procuring agency, most commonly on an ASP.NET MVC or .NET Core stack backed by MS-SQL Server, with the majority of functional modules built new for the engagement.

Agencies including Orange County Transportation Authority and Riverside County Transportation Commission (OCTA/RCTC), North Texas Tollway Authority (NTTA), and Central Texas Regional Mobility Authority (CTRMA) illustrate a pattern common across the tolling industry: functional requirements for trip processing, violation handling, customer account management, finance posting, inventory management, and reporting recur across engagements with substantial overlap, while agency-specific rules for pricing tiers, grace periods, dispute workflows, and interoperability with regional tolling networks vary. Treating every engagement as a full custom build extends the interval between contract award and operational go-live, increases gap-analysis effort during each request-for-proposal cycle, and leaves agencies maintaining functionally similar but structurally divergent platforms. The alternative pattern, in which a system is structured as a suite of small, independently deployable services organized around business capability rather than assembled as a single deployment unit, were articulated early in the shift away from monolithic enterprise design and offers a directly applicable response to this redundancy [1, 2]. A standardized, reusable architecture for tolling back-office functions responds directly to this pattern.

This composition presents a standardized microservices framework purpose-built for tolling back-office functions. Section 2 examines the decomposition of common BOS capabilities into discrete, independently deployable services and the domain boundaries that separate them. Section 3 sets out a reference architecture and technology stack, including the role of a common data model, a shared authentication and authorization layer, and containerized deployment on .NET Core. Section 4 addresses governance, security, and the mechanism by which agency-specific business rules attach to the shared core without altering it. Section 5 lays out a migration roadmap that carries legacy monolithic platforms toward the standardized framework in a controlled, agency-by-agency sequence. Together, these sections describe a path by which a tolling systems integrator moves from rebuilding entire platforms per procurement to configuring a shared, governed core.

2. Domain Decomposition and Service Design

Domain decomposition begins with separating the tolling back office into bounded functional areas that each own a distinct set of business capabilities and data. Trip transaction processing ingests toll-point events, associates them with a transponder or license-plate match, applies the correct toll rate, and produces a billable transaction record. Violation and pay-by-mail handling takes unmatched or unpaid transactions, applies statutory grace periods, issues notices, and manages the escalation and dispute lifecycle. Customer account management maintains subscriber profiles, prepaid and postpaid balances, payment instruments, and account status. Finance posting records toll revenue, fees, and adjustments against agency ledgers and reconciles them with payment processors and banking interfaces. Inventory and tag management tracks transponder issuance, replacement, and deactivation across vehicle classes. Reporting aggregates operational and financial data for agency, auditor, and regional interoperability consumption.

Separating these domains into independently deployable services, rather than modules within a single monolithic codebase, follows early descriptions of the microservice style, in which each service is built around a single business capability, owns its own data, and communicates with peers through a defined interface rather than a shared codebase [3]. A trip transaction service, for example, owns the transactional data associated with a toll event and exposes that data to other services exclusively through a defined interface, rather than through shared database tables. This boundary allows the violation-handling service to consume finalized trip records without holding direct write access to trip-processing data, and it allows the finance-posting service to subscribe to settled transactions without embedding toll-rate logic that belongs to trip processing. Early comparative evaluations of monolithic and microservice deployment patterns identify this data-ownership alignment as a primary determinant of long-term maintainability, since services organized around shared database access tend to accumulate coupling that reintroduces monolithic behavior inside a distributed system [4].

For a tolling back office, three decomposition principles keep the resulting service set both cohesive and independently maintainable. First, each service exposes a RESTful API as its only external interface, with request and response contracts published and versioned centrally so that agency-specific consumers and shared services evolve independently. Second, each service maintains its own persistence store, following a database-per-service pattern that prevents schema changes in one domain from propagating

uncontrolled effects into another. Third, cross-domain workflows, such as a violation notice that ultimately produces a finance posting, are coordinated through asynchronous events rather than synchronous chains of direct calls, which limits the operational blast radius when a single service experiences degraded performance or an outage.

Applying these principles consistently across trip processing, violation handling, account management, finance posting, inventory management, and reporting produces a service inventory that a systems integrator can reuse, with only configuration and rule-module changes, across successive agency engagements. The categories in Table 1 summarize the primary domain services and the decomposition considerations that shape each boundary.

Table 1. Core Domain Services and their Decomposition Boundaries [3, 4]

Domain Servicee	**Primary Responsibility**	**Data Ownership Boundary**	**Key External Interface**
Trip Transaction Processing	Match and rate toll-point events	Trip and transaction records	Trip Events API
Violation / Pay-by-Mail	Notice issuance and dispute lifecycle	Violation case records	Violation Case API
Customer Account Management	Subscriber profile and balance handling	Account and payment-instrument records	Account API
Finance Posting	Ledger entries and reconciliation	Financial transaction records	Ledger API
Inventory / Tag Management	Transponder issuance and lifecycle	Tag inventory records	Inventory API

3. Reference Architecture and Technology Stack

The reference architecture for a standardized tolling back office organizes the framework in to six layers, each addressing a distinct architectural concern. At the top, agency-specific configuration modules capture the business rules, fee schedules, notice templates, and workflow variations that distinguish OCTA/RCTC, NTTA, CTRMA, and comparable agencies from one another. These modules attach to the shared core through configuration and rule-engine extension points rather than through source-level modification, so a change requested by one agency does not require redeploying or retesting the shared services used by others. Directly beneath this layer, an API gateway provides the single entry point for all external and inter-agency traffic, handling routing, rate limiting, protocol mediation, and request aggregation before forwarding calls to the appropriate domain service, a structural role identified in early mapping studies of microservice architecture as a recurring element across independently developed implementations [5].

The core domain microservices layer hosts the six functional services described in Section 2, each built on .NET Core and packaged as an independently deployable container. Selecting .NET Core as the common runtime preserves continuity with the ASP.NET MVC and .NET Core expertise already applied across prior agency engagements, while gaining the platform's cross-platform hosting and container support. Beneath the domain layer, shared platform services supply capabilities every domain service requires but none should reimplement independently: identity and access management, centralized logging and monitoring, a service registry for inter-service discovery, and a configuration store for environment-specific settings. Centralizing these cross-cutting concerns follows architectural patterns documented in early systematic studies of microservice-based systems, where shared platform-level services recur across independently developed implementations as the layer most responsible for keeping a distributed, multi-team deployment maintainable at scale [6].

Persistence follows a common data model layered over database-per-service storage, in which each domain service owns its schema while agency-neutral entities, such as vehicle classification codes or toll-facility identifiers, are defined once in a shared reference model and referenced rather than duplicated across services. MS-SQL Server remains a supported persistence target, consistent with agency procurement standards and prior platform experience, while the architecture does not preclude cloud-native data stores where an agency's hosting environment favors them. The base of the architecture is the containerized deployment layer,

which packages every service, its dependencies, and its configuration into a portable unit, orchestrated through a continuous integration and delivery pipeline that supports consistent build, test, and release procedures across every agency environment, whether hosted on a private data center, a public cloud, or a hybrid combination of the two.

Figure 1 depicts the complete layered arrangement, from agency-specific configuration at the top through containerized deployment at the base, with the core domain microservices layer positioned as the shared functional center that every agency engagement reuses without rebuilding. It summarizes the architectural layers and their governing concern (Figure 1).

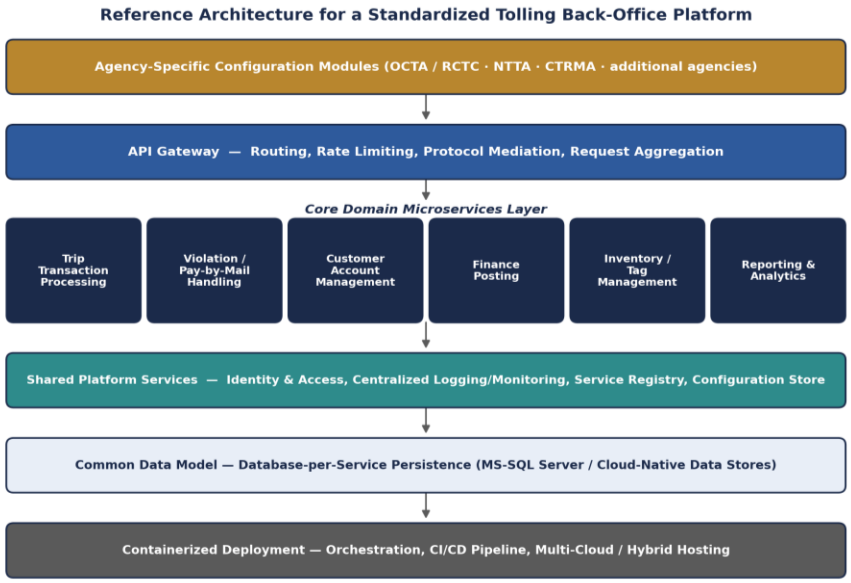

Figure 1. Layered Reference Architecture for the Standardized Tolling Back-Office Platform [5, 6]

4. Governance, Security and Multi-Tenant Configuration

Governance for a shared microservices framework establishes the standards by which every agency-specific engagement extends the common core without fragmenting it. A service contract template defines how new or modified RESTful endpoints are documented, versioned, and reviewed before a domain service exposes them, ensuring that a change made to satisfy one agency's requirement does not silently break compatibility for another agency consuming the same service. Governance also fixes the boundary between what belongs in the shared core and what belongs in an agency-specific configuration module: business logic that applies universally, such as the mechanics of matching a toll-point event to a transponder record, remains in the core service, while agency-particular policy, such as a specific grace-period duration or fee tier, remains external and configurable. Maintaining this boundary consistently is what allows the framework to scale across additional agency engagements without each one accumulating core-service modifications.

Security governance centers on a shared authentication and authorization service that every domain service and every agency-facing client consumes rather than reimplements. Centralizing identity verification at the API gateway, with token-based authorization propagated to downstream services, keeps credential handling in one auditable location and removes the need for each of the six domain services to manage its own authentication logic independently, an approach consistent with early guidance describing consistent, centrally enforced service contracts as a core tenet of sound microservice design [7]. Authorization decisions extend beyond authentication to service-to-service calls: a request forwarded from the violation-handling service to the finance-posting service carries a signed token that identifies both the originating service and the underlying agency context, allowing the receiving service to apply agency-specific access rules without maintaining its own copy of agency policy data. Early treatments of the microservice style note that decomposing a system into many independently deployed services shifts the security perimeter from a single boundary to numerous inter-service boundaries, a structural change that must be offset by consistent enforcement at the gateway together with defense-in-depth controls, such as input validation and encrypted transport, applied uniformly across every domain service [8].

Unified logging and monitoring completes the governance layer by giving agency operations staff and the systems integrator a single, consistent view across every domain service and every agency deployment. Centralized log aggregation and distributed tracing allow a support engineer to follow a single trip transaction from ingestion through violation handling and finance posting, even though the underlying request touches three separately deployed services. Consistent monitoring also supports the service-level agreements that agencies typically require of their tolling back office, since availability, latency, and error-rate metrics are captured in the same format regardless of which agency's traffic generated them, which in turn simplifies the audit and compliance reporting each agency must produce for its governing board.

Multi-tenant configuration governance, finally, determines how the framework isolates one agency's data and rules from another's while still sharing the same deployed services where appropriate. An agency may be hosted on a fully dedicated set of service instances, on a logically isolated tenant within shared instances, or on a hybrid arrangement in which high-sensitivity domains such as finance posting run dedicated while lower-sensitivity domains such as reporting run shared. The governance model documents which isolation pattern applies to each domain service and agency tier, and Table 2 summarizes the primary governance categories that structure this decision.

Table 2. Governance Categories and Control Mechanisms [7, 8]

Governance Category	Primary Control Mechanism	Applies To
Service Contract Standards	Versioned API template and review process	All domain services
Identity and Access	Centralized authentication and token-based authorization	Gateway and inter-service calls
Logging and Monitoring	Centralized aggregation and distributed tracing	All domain services and gateway
Tenant Isolation	Dedicated, shared, or hybrid hosting pattern	Agency-specific deployments
Configuration Boundary	Core-versus-configuration classification rule	Agency rule modules

5. Migration Roadmap from Monolithic to Microservices

Carrying an existing agency from a monolithic ASP.NET MVC or .NET Core back office toward the standardized microservices framework requires a controlled, incremental path rather than a single wholesale replacement. A full rewrite undertaken in one release exposes an agency to an extended period without a stable production system and concentrates risk at a single cutover event. An incremental strangler-fig pattern, in which new services progressively take over responsibilities from the legacy platform while the legacy system continues operating, has been documented as an effective route for migrating monolithic systems toward a cloud-native, independently deployable architecture without an extended service interruption [9]. Applying this pattern to a tolling BOS means the legacy monolith continues processing the domains not yet migrated while the API gateway routes traffic for migrated domains to their new microservice equivalents.

The roadmap proceeds through six phases. The assessment and domain-mapping phase inventories the legacy platform's modules and aligns them to the target domain boundaries defined in Section 2, identifying which legacy components correspond to trip processing, violation handling, account management, finance posting, inventory management, and reporting. The strangler-fig decomposition phase extracts the highest-priority domain, typically trip transaction processing given its position at the start of the transaction lifecycle, into an independently deployed service, with the API gateway directing new traffic to it while the legacy platform continues serving domains not yet migrated. The parallel-run validation phase operates the newly extracted service alongside its legacy counterpart, reconciling output between the two to confirm that rate calculations, transaction volumes, and downstream records match before the legacy component is retired for that domain, mirroring the comparative deployment evaluations used in early studies to validate a service pattern before full adoption [10].

The agency cutover phase shifts an agency's live traffic for a given domain fully onto the new service once parallel-run validation confirms output parity, and this phase repeats domain by domain until every functional area operates on the standardized

framework. The legacy decommission phase retires the corresponding monolithic components once no agency traffic depends on them, consolidates any remaining data into the common data model, and removes the infrastructure supporting the retired modules. The continuous governance phase, which persists after the migration itself concludes, applies the governance categories established in Section 4 to keep the reference architecture, service contracts, and roadmap current as additional agencies join the shared platform or as existing agencies request new configuration modules.

A defining property of this roadmap is that each phase applies independently to each agency's domain migration, so OCTA/RCTC, NTTA, CTRMA, and any subsequent agency can progress on separate timelines while all of them converge on the same governed reference architecture. Figure 2 illustrates the six-phase sequence and the domain-by-domain independence that allows staggered agency timelines to share one governance framework. Figure 2 summarizes the migration phases and their principal risk-control mechanism.

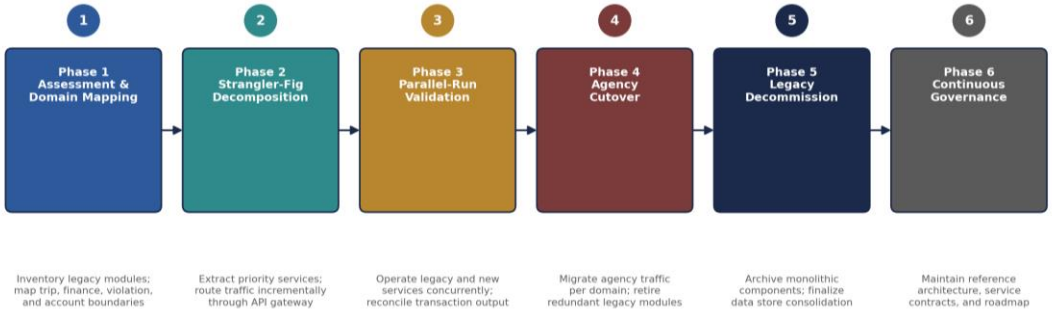

Figure 2. Six-Phase Migration Roadmap from Monolithic Back-Office Platforms to the Standardized Framework [9, 10]

6. Conclusion

A standardized microservices framework for tolling back-office functions replaces repeated, agency-specific ground-up development with a shared, governed core that each engagement configures rather than rebuilds. Decomposing trip transaction processing, violation and pay-by-mail handling, customer account management, finance posting, inventory and tag management, and reporting into independently deployable services, each with a defined RESTful contract and its own data ownership boundary, keeps the shared core stable while leaving room for the agency-specific policy differences that distinguish OCTA/RCTC, NTTA, and CTRMA-style engagements. A layered reference architecture, anchored by a shared API gateway, common platform services for identity and monitoring, and a containerized deployment model on .NET Core, gives every agency engagement a consistent technical foundation. Governance standards for service contracts, security, and tenant isolation keep that foundation coherent as additional agencies and configuration modules join the platform over time. An incremental, strangler-fig migration roadmap lets an existing monolithic agency platform move toward the standardized framework domain by domain, validating output at each step rather than committing to a single high-risk cutover. Together, these elements shorten the interval between contract award and operational deployment, reduce the gap-analysis burden that each new procurement otherwise carries, and give a tolling systems integrator a governed, extensible platform that scales across an expanding portfolio of agency clients. The direct implication for organizations delivering these systems is that the standardization effort itself becomes a durable asset: each additional agency engagement strengthens the shared core rather than duplicating it, and the accumulated configuration library becomes a competitive and operational advantage that a series of one-off custom builds cannot replicate.

References

[1] Lewis, J., & Fowler, M. (2014). Microservices: A definition of this new architectural term. MartinFowler.com. https://martinfowler.com/articles/microservices.html
[2] Newman, S. (2015). Building microservices: Designing fine-grained systems. O'Reilly Media.
[3] Thönes, J. (2015). Microservices. IEEE Software, 32(1), 116. https://doi.org/10.1109/MS.2015.11
[4] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th Computing Colombian Conference (10CCC) (pp. 583–590). IEEE. https://doi.org/10.1109/ColumbianCC.2015.7333476

[5] Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. In 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 44–51). IEEE. https://doi.org/10.1109/SOCA.2016.15

[6] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018) (pp. 221–232). SciTePress. https://doi.org/10.5220/0006798302210232

[7] Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. Computer Science – Research and Development, 32(3–4), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[8] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), Present and ulterior software engineering (pp. 195–216). Springer. https://doi.org/10.1007/978-3-319-67425-4_12

[9] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software, 33(3), 42–52. https://doi.org/10.1109/MS.2016.64

[10] Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (pp. 137–146). SciTePress.