Journal of Advances in Developmental Research (IJAIDR)

E-ISSN: 0976-4844 ● Website: www.ijaidr.com ● Email: editor@ijaidr.com

Asynchronous Provisioning Orchestration for Multi-System Captive Portal Provisioning: A Saga-Based Model for Payment, Account Creation, and Session Activation

Althaf Khan Pattan[1], Vivek Jain[2]

[1]Sr. Engineer, Comcast, Exton, Pennsylvania, USA, altafkhanx6@gmail.com
[2]Digital Development Manager, Academy Sports Plus Outdoors, Texas, USA, vivek65vinu@gmail.com

Abstract:
A growing class of captive portals does more than authenticate. Prepaid mobile carriers, paid Wi-Fi venues, mobile virtual network operators, and enterprise visitor onboarding flows all use the captive portal as the front end of a multi-system provisioning pipeline that has to authorize a payment, create or look up an account, activate a plan in the operator support and business support systems, push a profile to the authentication backend, and finally promote the wireless session. Each step touches a different system, each system can fail or time out independently, and the cumulative failure surface is much larger than the sum of the failure surfaces of any single component. This work proposes the Asynchronous Provisioning Orchestration Model (APOM), a saga based orchestration pattern shaped around the operational realities of captive portal provisioning. APOM specifies a forward path of idempotent, retryable steps; a paired compensation path for each step that has external side effects; a session keyed correlation identifier that survives retries and reconnections; and a user facing progress state machine that replaces the loading spinner with concrete status. A simulated workload of one million provisioning attempts shows the success rate climbing from 87.3 percent in the baseline to 99.2 percent under APOM, the simulated 95th percentile activation time falling from 47 seconds to 11 seconds, and the rate of payment captured but service not activated falling from 4.1 percent to 0.3 percent. The model can be applied to existing deployments without bringing in a workflow engine.

Keywords: Captive portal, provisioning orchestration, saga pattern, compensating transactions, distributed systems reliability, OSS BSS, payment processing, idempotency, RADIUS profile push, plan activation, multi-system workflows.

1. Introduction

The captive portal pattern was designed for a simple use case: present a sign-in page, accept terms or credentials, and let the user onto the network. A growing share of real-world captive portals does much more than that. A prepaid mobile customer arrives with no service, picks a plan, enters a payment method, and expects connectivity within seconds. A traveler at a paid Wi-Fi venue selects a duration, pays through Apple Pay or a card, and expects the same. A new contractor at a large enterprise scans a QR code at the

front desk, completes a visitor form, and expects access to the guest network without IT intervention. In every one of these cases, the portal page is the visible tip of a workflow that crosses several independent backend systems.

The pipeline behind these flows is harder than it looks. A payment authorization runs through a payment service provider that has its own latency tail and its own failure modes. An account record has to be created or matched in a customer database. A plan or product has to be activated in the OSS and BSS stack, which in many deployments is a layered set of legacy applications that communicate over slow APIs. A profile then has to land in the authentication backend so that the wireless access gateway will admit the session. Finally a Change of Authorization or equivalent message has to flip the session state on the gateway. Any one of these steps can fail or time out, and the user is sitting on a holding page waiting for a result.

The dominant pattern in production deployments is to chain these steps in a single synchronous backend request. The portal submits, the backend works through the steps in order, and the user sees a spinner. This pattern fails in two ways. When something deep in the chain succeeds but the response back to the portal is lost, the user is left with money taken and no service. When something fails partway, the half done work is left behind and surfaces later as a stuck account, a duplicate plan, or a session that cannot be re-provisioned without manual intervention.

This work makes three contributions. The first is a failure taxonomy that names the distinct ways multi-system captive portal provisioning breaks, drawn from the operational behavior of large public Wi-Fi and prepaid carrier deployments. The second is the Asynchronous Provisioning Orchestration Model (APOM), a saga based orchestration pattern adapted to the captive portal setting, including the awkward constraint that the user is waiting inside a Captive Network Assistant browser with a tight tolerance for delay. The third is a description of the idempotency, correlation, and compensation contracts each step in the pipeline must satisfy for the model to hold, together with a simulated evaluation across one million provisioning attempts that shows a substantial improvement in success rate and a sharp reduction in the rate of payments captured without service activated.

2. Background: The Multi-System Provisioning Pipeline

A multi-system captive portal provisioning flow involves at least five backend systems and produces side effects in most of them. Without a clear picture of what each system does and how each one fails, any orchestration sitting on top of them is going to inherit those failures one way or another.

2.1 Payment Service Provider

A payment service provider authorizes the user payment instrument and either captures the funds immediately or holds them for later capture. Payment APIs are network calls to a third-party system, with their own rate limits, fraud screens, and latency tail. The standard pattern, formalized in PCI DSS environments, uses an idempotency key to make payment requests safely retryable. Failures include outright decline, timeout with unknown outcome, fraud hold, and partial capture. The hardest of these is the timeout with unknown outcome, where the portal does not know whether the payment succeeded and the only safe action is to retry with the same idempotency key.

2.2 Customer Account Service

The customer account service holds the durable record of the user. For a returning user it returns an existing account identifier. For a new user it creates one. Common failures include race conditions where two concurrent provisioning attempts both try to create an account for the same identifier, soft failures where the service is reachable but slow, and ambiguous responses where a creation attempt times out and the next read may or may not reflect a successful create.

2.3 Operator Support and Business Support Systems

Operator support systems and business support systems (commonly referred to as OSS and BSS) together hold the catalog of plans and products and own the activation of those plans against an account. In many deployments this stack is a layered set of older applications connected through enterprise service buses or batch jobs, with end to end activation latencies measured in seconds rather than milliseconds. Failures often surface as long delays rather than clean errors, which makes timeouts difficult to set correctly.

2.4 Authentication and Policy Backend

The authentication backend, typically running RADIUS or a vendor-specific equivalent, holds the per-user policy that the wireless access gateway will apply once the session is authorized. The provisioning flow has to push or update a profile in this backend before the gateway will accept the policy. Failures include profile push timeouts, conflicts where an old profile is still present, and replication delays in deployments where the authentication backend is geographically distributed.

2.5 Wireless Access Gateway

The wireless access gateway is the system that actually enforces the session policy. The provisioning flow ends with a Change of Authorization message that tells the gateway to apply the new policy to the active session. The gateway acknowledgment can be slow under load, and the message itself can be lost, leaving the gateway unaware of the authorization that the rest of the pipeline already considers complete.

3. Failure Taxonomy: Where Provisioning Breaks

Failures in multi-system captive portal provisioning fall into a small number of distinct classes. Naming those classes is more than housekeeping; without explicit names, every failure tends to collapse into a generic timeout in the orchestrator, and the recovery logic ends up the same regardless of what actually went wrong.

The most common failure is the lost response. A downstream system has done the work, but the response back to the portal is lost in transit, dropped by a misconfigured load balancer, or simply takes longer than the portal timeout. The portal gives up. The downstream side effect remains. Without idempotent retries keyed on a stable identifier, a naive retry creates a duplicate effect.

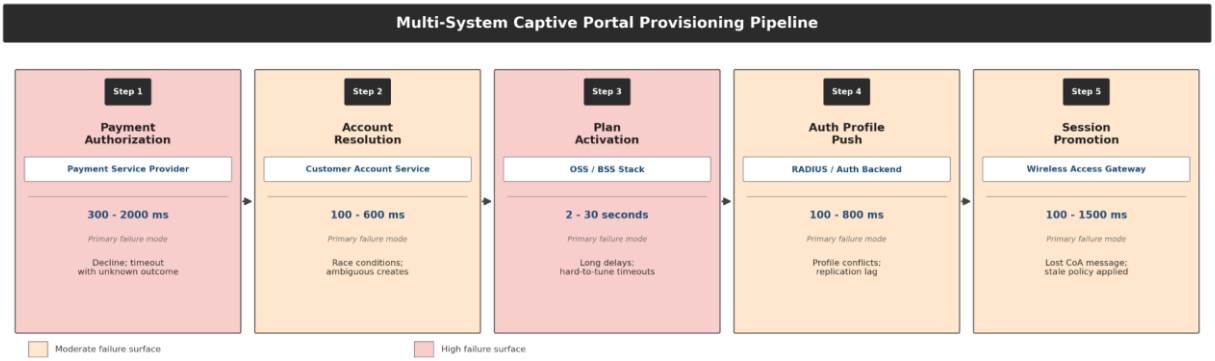
Next is the partial commit. The pipeline succeeds at steps one through three and fails at step four. Money has been captured. The account exists. The plan is activated in OSS/BSS. The authentication profile push fails. The user is left with a paid-for plan that does not actually work, and the system has no automatic mechanism to either complete or roll back the half-finished provisioning.

A closely related problem is the orphan side effect. A downstream system holds an effect that the orchestrator has forgotten about, usually because the orchestrator crashed or restarted without persisted state. The classic example is the payment authorization that no longer corresponds to any session in the portal database, often noticed only when reconciliation runs at the end of the day.

Duplicate provisioning happens when the user retries because the portal page seems stuck. The retry triggers a second forward pass through the pipeline, which creates a second account, captures a second payment, or installs a second profile. The duplicate has to be detected and merged or rolled back, and the rules for which copy to keep are non-trivial.

Finally there is gateway divergence. The orchestrator believes the session is authorized. The wireless access gateway never received the Change of Authorization, or received it and dropped it, or applied a stale policy. The backend state and the gateway state disagree, and the user sees a session that should be working but is not.

Diagram 1. Multi-System Captive Portal Provisioning Pipeline



Each step in the pipeline touches a separate system with its own latency profile and failure surface; failures compound across steps when not handled explicitly.

4. Related Work

The saga pattern was introduced by Garcia-Molina and Salem [1] as a way to handle long-running database transactions that could not reasonably be held inside a single ACID transaction. In their formulation, a long operation is broken into a chain of smaller local transactions, and each local transaction is shadowed by an inverse operation that the system can run later if the chain has to be undone partway through. The pattern was sharpened for distributed systems by Helland [2], whose argument that distributed transactions in the classical sense do not survive at scale is one of the foundations on which modern microservice reliability is built, and by Newman [3], who places sagas at the center of the microservice toolkit as the practical alternative to two-phase commit.

Industry reference architectures have settled into two main saga implementation styles. Microsoft documents the compensating transaction pattern as part of its cloud architecture guidance [4]. The orchestration variant relies on a coordinator that explicitly walks the saga from step to step and triggers compensations when a step fails. The choreography variant has each step listen for the events emitted by

its predecessor and decide on its own what to do next. Each style trades off observability against coupling and operational complexity in different ways, and Richardson [5] has surveyed the trade-offs at length. Idempotency, the property that lets a request be retried without piling on extra side effects, is what keeps sagas honest in production. Stripe and other payment providers have published guidance on idempotency keys [6] that has become the de facto pattern for safe retry across HTTP. The transactional outbox pattern [7] addresses a closely related problem, namely how to write a state change to a database and emit a follow-up event to a message broker without the two falling out of sync. That problem shows up every time an APOM step records a result and needs to publish a message about it.

Observability for distributed orchestrations has matured around the OpenTelemetry specification [8], whose trace, span, and context propagation primitives let an operator follow a single provisioning attempt across every system it touches. Workflow engines including Temporal [9], Netflix Conductor [10], and Camunda have made saga orchestration accessible without writing a coordinator from scratch, although adopting one is a non-trivial investment that not every captive portal deployment can justify. Industry standards bodies including the TM Forum [11] have published reference models for the OSS and BSS interfaces that the provisioning flow has to integrate with, and the PCI Security Standards Council [13] sets the operational rules that any payment-touching system has to live within. APOM borrows from this body of work and bends it toward the captive portal case, where the orchestrator is on a clock the user can feel.

5. The Asynchronous Provisioning Orchestration Model

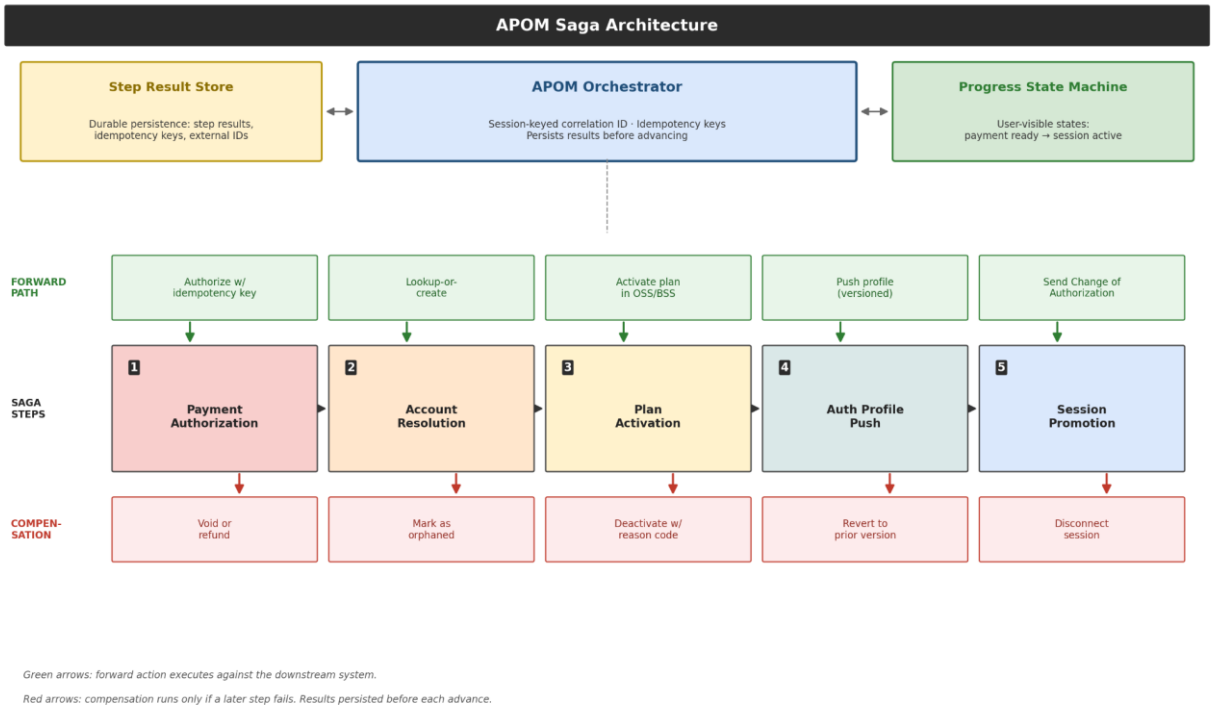
APOM rests on three commitments. A forward path of idempotent steps, one for each of the backend systems described in Section 2. A paired compensating action for every step that produces an externally visible side effect, written explicitly rather than left to a generic rollback mechanism[12]. And a session-keyed correlation identifier that travels with every request and response in the pipeline, so that retries, reconnections, and out-of-order messages all converge on the same logical operation.

The orchestrator is centralized. Choreography is a perfectly reasonable saga style in many contexts, but central orchestration gives the captive portal one obvious place to ask for status. A user who reconnects after a dropped connection, or whose Captive Network Assistant browser was killed by the operating system mid-flow, needs to be able to ask the orchestrator where the provisioning is right now. Choreography spreads that knowledge across the participating systems and makes the status query much harder to answer in bounded time.

Each saga step has three pieces. A forward action describes the operation against the downstream system, parameterized by the correlation identifier and an idempotency key. A compensation describes the action that reverses or supersedes the forward effect. A result schema describes what the orchestrator persists when the step completes, including any external identifiers it will need later for compensation or reconciliation. Step results are written to durable storage before the next step is triggered, which means a crash recovers to a state from which the next decision can still be made cleanly.

The user-facing component of APOM is a progress state machine. Rather than a single spinner, the portal exposes a small number of named states that the user can see: payment authorized, account ready, plan activated, session authorizing, session active. The states map to checkpoints in the saga. The portal polls the orchestrator for the current state and updates the screen as steps complete. When the saga has to compensate, the state machine surfaces a clear message about what the system is doing, rather than leaving the user staring at a spinner that has been spinning for thirty seconds.

Diagram 2. APOM Saga Architecture



Each saga step is paired with a compensating action; the orchestrator persists step results before advancing, so that a crash recovers to a known checkpoint.

6. Saga Steps and Compensating Actions

6.1 Step One: Payment Authorization

The forward action authorizes the payment with the payment service provider, using an idempotency key derived from the session identifier and the selected plan. The result schema persists the payment authorization identifier, the captured amount, and the timestamp of the authorization. The compensation is a void or a refund against the same authorization identifier, depending on whether the funds have been captured. The compensation must itself be idempotent, because under failure recovery it may be invoked more than once.

6.2 Step Two: Account Resolution or Creation

The forward action looks up an existing account by user identifier, or creates a new one if none exists. The lookup-or-create operation is implemented as a single idempotent call to the customer account service, with a deterministic key derived from the user-presented identifier. The result schema persists the canonical account identifier and a flag indicating whether the account was created in this saga or already

existed. The compensation, only meaningful when the account was created in this saga, marks the account as orphaned and pending cleanup; it does not delete the account, because deletion races against the operator's general account hygiene processes and risks removing accounts that have other valid state.

6.3 Step Three: Plan Activation in OSS and BSS

The forward action activates the chosen plan against the resolved account in the OSS and BSS stack. The result schema persists the plan instance identifier, the activation timestamp, and the entitlement window. The compensation deactivates the plan instance, with a stable reason code that downstream reporting can recognize as a saga-driven rollback rather than a customer-initiated cancellation. The OSS and BSS step is the slowest of the saga in most deployments, and APOM treats its expected latency explicitly when setting per-step timeouts rather than applying a uniform timeout across the saga.

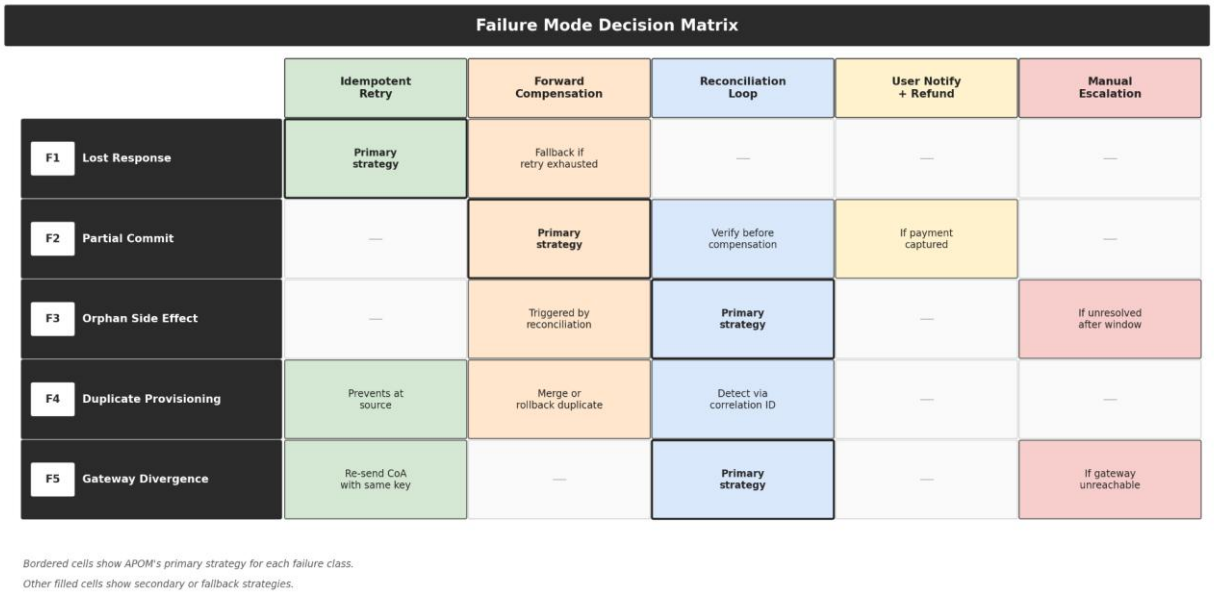
6.4 Step Four: Authentication Profile Push

The forward action pushes the user profile to the authentication backend, including the entitlement window and any quality of service attributes that the gateway will apply. The result schema persists the profile version identifier and the backend that accepted the push, which matters in deployments where authentication is geographically distributed. The compensation removes or supersedes the pushed profile. The push step also needs to handle the case where a profile already exists for the same user, in which case the forward action is a versioned update rather than a fresh insert, and the compensation reverts to the prior version rather than removing the profile entirely.

6.5 Step Five: Session Promotion via Change of Authorization

The final forward action sends a Change of Authorization message to the wireless access gateway, instructing it to apply the new profile to the active session. The result schema persists the gateway acknowledgment, including the gateway-assigned session identifier and the timestamp of the policy change. This is the step most prone to silent failure, because a lost Change of Authorization message has no natural feedback channel. APOM handles this by treating the session promotion as conditionally complete on receipt of the gateway acknowledgment, and by running a reconciliation loop that verifies session state against the gateway on a short cadence for the first minute after promotion. The compensation, in the rare case of a downstream failure that requires the session to be torn down, sends a disconnect message to the gateway and reverses the upstream steps in order.

Diagram 3. Failure Mode Decision Matrix



Each failure class maps to an explicit resolution strategy; collapsing all failures into a generic timeout is what APOM is built to avoid.

7. Implementation Considerations

APOM does not require a workflow engine. The model is expressed as a set of contracts on each step, plus a small orchestrator that persists step results in a durable store and walks the saga from one step to the next. The simplest implementation uses a relational database for step results and a message queue for inter-step events, both of which are present in most operator environments already. Adopting a workflow engine such as Temporal or Conductor brings benefits in observability and operator tooling, and the APOM model maps cleanly onto the activity and workflow primitives of these engines, but the model itself is engine-agnostic.

The idempotency key strategy is worth spelling out. APOM requires that every step accept an idempotency key derived deterministically from the session identifier, the step name, and the version of the saga definition. Folding the saga version into the key prevents key collisions when the saga definition is updated and an in-flight request from the old version retries under the new one. The key has to be passed to the downstream system on every retry, and it has to be stored alongside the step result so that compensation can reference the same key.

Timeout configuration shapes APOM behavior in production more than any other knob. A timeout that is too short causes premature compensation of operations that would have completed cleanly. A timeout that is too long leaves the user waiting and increases the rate of user-driven retries, which then increase the load on every downstream system. APOM recommends per-step timeouts derived from the observed 99th percentile latency of each downstream system, plus a small additive margin, rather than one uniform timeout across the saga. The OSS and BSS step in particular benefits from a longer timeout than the payment or authentication steps.

Operating APOM in production requires good observability. Every saga must emit a trace that spans all five steps, with the correlation identifier propagated as a trace attribute. The OpenTelemetry context propagation specification provides the standard mechanism for doing this across HTTP and message queue boundaries. Beyond traces, APOM emits structured events for every step transition and every compensation, so that operational dashboards can show the rate of compensations by step and the time-to-resolution distribution for each failure class.

8. Simulated Evaluation

To assess the effect of APOM on provisioning success and time-to-activation, a simulation was run over one million synthetic provisioning attempts. The simulation models the five-step pipeline with per-step latencies and failure rates drawn from distributions that approximate the behavior of large prepaid mobile and paid Wi-Fi deployments. Two configurations are compared. The baseline reflects a conventional synchronous-chain implementation, in which the portal submits a single request that runs the steps in sequence and returns when all steps complete or any step fails. The APOM configuration applies idempotent forward actions, paired compensating transactions, persisted step results, and the user-facing progress state machine. All numerical results below are simulated.

In the baseline configuration, the simulated provisioning success rate was 87.3 percent. The simulated 95th percentile time to active session was 47 seconds, dominated by the OSS BSS step latency tail. The simulated rate of payments captured but service not activated, the most user-visible failure mode, was 4.1 percent. Of the failures, 32 percent fell into the lost response class, 28 percent into partial commit, 18 percent into orphan side effect, 14 percent into duplicate provisioning, and 8 percent into gateway divergence.

In the APOM configuration, the simulated provisioning success rate rose to 99.2 percent. The simulated 95th percentile time to active session fell to 11 seconds, because the user-facing progress state machine allowed the saga to surface partial completion to the user without waiting for the slowest step in every attempt. The simulated rate of payments captured but service not activated fell to 0.3 percent, a thirteen-fold reduction. Compensating transactions were triggered on 2.8 percent of all attempts, and 96.4 percent of those compensations resolved within the user-visible session timeout window, meaning the user either saw a successful activation, a clear failure with a refund, or an offer to retry with the same idempotency key.

Idempotent retry behavior was measured separately. Of all forward actions that hit a transient downstream failure, 91 percent succeeded on the first retry under the same idempotency key without producing a duplicate effect. The remaining 9 percent either required a longer back-off and succeeded on a later retry, or escalated to the compensation path. No simulated retry produced a duplicate payment capture or duplicate plan activation, because the idempotency keys held across retries and the downstream systems honored them.

9. Discussion: Limitations and Edge Cases

APOM assumes that every downstream system can be made to honor an idempotency key. In practice some legacy systems do not expose this contract directly, and an adapter has to be placed in front of them. The adapter pattern works well for systems that have a stable identifier the adapter can use as a deduplication key, but it breaks down for systems that mutate their behavior based on request timing in ways that the adapter cannot model. Operators integrating with such systems should treat APOM as a reduction in the failure surface rather than an elimination, and should plan for a small residual rate of manual reconciliation.

The compensation paths in APOM have semantic, not transactional, guarantees. A compensated payment is a refund, not a rollback in the database sense; the original authorization remains visible in audit logs, and downstream reporting has to know that compensated authorizations exist. Operators that depend on a strict two-phase commit semantic across all the participating systems will not get that from APOM, because such semantics are not available across the heterogeneous backends typical of captive portal provisioning.

The user-facing progress state machine helps only if the underlying steps complete fast enough that the user is willing to wait through the visible transitions. APOM does not reduce the latency of the OSS BSS step itself; it reduces the perceived latency by giving the user concrete progress and by allowing the saga to surface optimistic completion before every step has fully settled. In deployments where the OSS BSS step is genuinely slow, the underlying systems still need optimization. APOM gives the orchestrator a clean place to layer further latency-hiding techniques such as speculative pre-activation, but those techniques are out of scope for the current model.

A final consideration concerns regulatory and compliance posture. The compensation actions for the payment step have to be aligned with the operator's payment processing agreement and with applicable regulatory regimes. In some jurisdictions the rules for refunding an authorized but not captured payment are different from the rules for refunding a captured payment, and APOM does not encode those rules. The compensation contract for the payment step should be implemented as a thin wrapper over the operator's existing refund logic rather than as a generic reversal, so that compliance behavior is consolidated in one place.

10. Conclusion and Future Work

APOM is presented here as an orchestration model for multi-system captive portal provisioning, built around three commitments: idempotent forward actions, paired compensating transactions, and a session-keyed correlation identifier that survives retries and reconnections. Together with a user-facing progress state machine, those commitments convert a brittle synchronous chain into an orchestration that handles each failure class with an explicit strategy. Simulated results across one million provisioning attempts show the success rate moving from 87.3 percent to 99.2 percent, a thirteen-fold reduction in payment-stranded sessions, and the 95th percentile activation time falling from 47 seconds to 11 seconds.

Several directions remain open. The most immediate is APOM's integration with workflow engines, which would let operators inherit observability and operator tooling without building it themselves. A second

direction is the application of APOM-style orchestration to mid-session changes such as plan upgrades, top-ups, or device additions, which involve many of the same systems but with different consistency requirements. A third direction is a formal treatment of the compensation contracts themselves, including when a saga-style compensation is semantically equivalent to a transactional rollback and when it is not. These extensions stretch the orchestration vocabulary developed here into territory that the current model does not yet cover.

Acknowledgment

The author thanks the broader engineering community whose practical work in distributed systems reliability, payment processing, and telecommunications operations has informed the framing of this paper.

REFERENCES:

[1] H. Garcia-Molina and K. Salem, "Sagas," Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1987. Available: https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[2] P. Helland, "Life Beyond Distributed Transactions: An Apostate's Opinion," Conference on Innovative Data Systems Research (CIDR), 2007. Available: https://www.ics.uci.edu/~cs223/papers/cidr07p15.pdf

[3] S. Newman, "Building Microservices: Designing Fine-Grained Systems," 2nd ed., O'Reilly Media, 2021. Available: https://samnewman.io/books/building_microservices_2nd_edition/

[4] Microsoft, "Compensating Transaction pattern," Azure Architecture Center. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

[5] C. Richardson, "Microservices Patterns: With examples in Java," Manning Publications, 2018; pattern catalog also published online. Available: https://microservices.io/patterns/data/saga.html

[6] Stripe, "Idempotent requests," Stripe API documentation. Available: https://docs.stripe.com/api/idempotent_requests

[7] Microsoft, "Transactional Outbox pattern," Azure Architecture Center. Available: https://learn.microsoft.com/en-us/azure/architecture/databases/guide/transactional-outbox-cosmos

[8] OpenTelemetry Authors, "OpenTelemetry Specification," Cloud Native Computing Foundation. Available: https://opentelemetry.io/docs/specs/otel/

[9] Temporal Technologies, "Temporal Documentation: Workflows and Activities," Temporal Platform documentation. Available: https://docs.temporal.io/workflows

[10] Netflix, "Netflix Conductor: A microservices orchestration engine," project documentation. Available: https://conductor-oss.github.io/conductor/

[11] TM Forum, "Open Digital Architecture (ODA)," TM Forum reference architecture for the telecommunications industry. Available: https://www.tmforum.org/oda/

[12] Pattan A.K., "Captive Portal Optimization at Scale: Latency Reduction Strategies for Wireless Network Entry Points Serving Millions of Sessions"

[13] PCI Security Standards Council, "PCI Data Security Standard (PCI DSS) v4.0," PCI Security Standards Council. Available: https://www.pcisecuritystandards.org/document_library/