IshemaLink Submission Reports

Author: Carine Umugabekazi

Course: Advanced Python Programming — ALU, January 2026

Github

Report 1: Integration Report

How IshemaLink Solved the Domestic vs. International Logic Conflict

1.1 The Problem

When designing IshemaLink, the central architectural question was how to handle domestic and international shipments without duplicating business logic. The two shipment types share the same lifecycle creation, tariff calculation, payment, driver assignment, and delivery, but differ in pricing rules, documentation requirements, and government compliance needs:

- **Domestic shipments** operate within Rwanda's borders, use a base tariff of 1,000 RWF per kilogram, and require RURA vehicle compliance verification.
- **International shipments** cross EAC borders, use a higher base tariff of 3,000 RWF per kilogram, require customs manifest generation in XML format, and involve border control documentation.

The naive solution, two separate Django apps with completely independent models, views, and payment flows, would have duplicated every piece of shared logic and made maintenance a constant source of inconsistency.

1.2 The Solution: One Unified Core Model

The design decision was to place a single `Shipment` model inside the `Core` app that serves both shipment types through **a shipment_type** discriminator field:

```python
class Shipment(models.Model):
    SHIPMENT_TYPE_CHOICES = [
        ("domestic", "Domestic"),
        ("international", "International"),
    ]
    shipment_type = models.CharField(max_length=20,
choices=SHIPMENT_TYPE_CHOICES)
    weight = models.FloatField()
    phone_number = models.CharField(max_length=20)
    status = models.CharField(max_length=30, default="pending_payment")
    tariff = models.FloatField(default=0)
    assigned_driver = models.ForeignKey(Driver, null=True, blank=True,
on_delete=models.SET_NULL)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

All shipments share the same database table, the same driver assignment logic, the same payment workflow, and the same status lifecycle: pending_payment to `confirmed to confirmed_no_driver or payment_failed. The Domestic and International Django apps exist as dedicated modules ready to house type-specific models and views as the platform grows, without requiring any changes to the Core model.

1.3 The BookingService: Where Integration is Orchestrated

The BookingService class is the single entry point for the complete booking workflow, regardless of shipment type:

Create Shipment → Calculate Tariff → Initiate MoMo Payment
→ Await Webhook Callback → Assign Driver → Send Notifications

The tariff calculation is the only step where domestic and international logic diverges:

```python
base_tariff = 1000 if shipment_type == "domestic" else 3000
tariff = base_tariff * weight
```

Every other step: database persistence, payment initiation, driver assignment, SMS and email notification is identical for both types. This means a bug fix or improvement to the booking flow applies universally, with no risk of the two shipment types drifting out of sync.

1.4 Dependency Injection for Clean Service Separation

BookingService receives its external dependencies through constructor injection:

```python
class BookingService:
    def __init__(self, payment_gateway: MomoMock, notifier: NotificationEngine):
        self.payment_gateway = payment_gateway
        self.notifier = notifier
```

This means the core booking logic is completely decoupled from the payment provider and notification channel. MomoMock can be swapped for a real MTN or Airtel SDK in production without touching a single line of BookingService. The same applies to the notification engine; switching channels requires no changes to the booking workflow itself.

1.5 ACID Compliance and Payment Failure Handling

All shipment creation is wrapped in @transaction.atomic, ensuring that if any step fails, the entire operation rolls back and no partial data is saved. The handle_payment_callback method implements explicit, granular status transitions:

- Payment success + driver available to confirmed, driver marked unavailable, SMS and email sent
- Payment success + no driver to confirmed_no_driver, payment preserved, operational gap flagged
- Payment failure to payment_failed, SMS sent to agent, driver availability unchanged

This granular status system ensures no financial record is ever silently lost, and the system always reflects the true state of each shipment.

1.6 Why This Architecture Scales

A single unified Shipment table means analytics queries across all shipment types are simple aggregations; no JOIN across separate tables is needed to answer "total revenue this week" or "most active districts." As MINICOM requests route analytics and revenue heatmaps through the BI endpoints, this unified structure makes those queries straightforward and fast.

Report 2: Scalability Plan

How IshemaLink Will Handle 50,000 Users in 2027

2.1 Current State (2026 Baseline)

IshemaLink currently runs on a single-server Docker Compose stack. The production configuration includes one Django/Gunicorn instance with 4 workers, one PostgreSQL database fronted by PgBouncer, one Redis instance, one Celery worker, and Nginx as a reverse proxy. This is the correct foundation — every component is already containerized and independently replaceable.

2.2 Phase 1: Vertical Scaling and Tuning (0–10,000 Users)

No code changes are required for this phase — only configuration tuning.

Gunicorn workers: On a production server with 8 CPU cores, workers increase from 4 to 17 using the formula (2 × cores) + 1, giving 17 parallel request handlers on the same machine.

PgBouncer: Already configured in production with MAX_CLIENT_CONN=1000 and DEFAULT_POOL_SIZE=20. This means 1,000 application connections share just 20 real PostgreSQL connections — critical for harvest season spikes.

Redis caching: The admin_dashboard_summary_view currently runs three separate database queries on every request. With a 60-second Redis cache on this view, the majority of dashboard requests become a single cache read with no database involvement.

Estimated capacity: 10,000 concurrent users.

2.3 Phase 2: Horizontal Scaling (10,000–30,000 Users)

Multiple app instances: Run 3–5 Django/Gunicorn containers behind Nginx upstream load balancing. Nginx already acts as the reverse proxy; adding upstream servers requires only a configuration change.

PostgreSQL read replica: The four analytics endpoints run read-only aggregation queries. Routing these to a read replica doubles read capacity while keeping all writes on the primary.

Celery scaling: Multiple Celery workers run in parallel, each consuming from the Redis queue independently. Payment callbacks and notifications scale horizontally with no application code changes.

Estimated capacity: 30,000 concurrent users.

2.4 Phase 3: Query Optimization (30,000–50,000 Users)

Materialized views: At 50,000 users, the Shipment table will contain millions of rows. Converting the analytics endpoint queries to PostgreSQL materialized views refreshed hourly, which means the BI API reads pre-computed results in milliseconds instead of scanning the full table on every request.

Full caching layer: Cache decorators on all read-heavy views. The tracking endpoint returns simulated coordinates today in production; last-known GPS coordinates are cached per shipment ID and updated only when the driver's device syncs, avoiding database reads on every polling request.

Estimated capacity: 50,000+ concurrent users.

2.5 Rwanda-Specific Considerations

Offline sync: Agents in Nyamagabe or Musanze may lose connectivity for hours during harvest season. The mobile app must queue shipment creation requests locally and retry. BookingService.create_shipment uses @transaction.atomic, which makes it safe to retry a failed transaction and leaves no partial data.

SMS priority: The NotificationEngine sends both SMS and email. In low-connectivity areas, email is unreliable. SMS must always be sent first, as it works on 2G and requires no smartphone. This is already the pattern in handle_payment_callback: SMS goes to shipment.phone_number immediately, email is secondary.

Data sovereignty: All infrastructure runs within Rwanda. The MinIO backup system is self-hosted. No data routes through foreign cloud providers for primary storage. As user volume grows, this constraint remains non-negotiable.

Report 3: Local Context Essay

Why Generic Logistics Software Fails in Rwanda, and How IshemaLink Succeeds

3.1 The Assumption Problem

Most logistics software is built on unstated assumptions: bank account ownership, reliable high-speed internet, standardized road infrastructure, and integration with Western regulatory systems. In Rwanda, these assumptions fail at every level, not because Rwanda is behind, but because Rwanda's digital economy developed along a different and more practical path.

The consequence is predictable. A generic platform imported into Rwanda asks agents to pay through systems requiring credit cards they do not own. Drivers receive app notifications that need constant mobile data on roads where 4G is unavailable. Compliance modules generate reports that RRA and RURA do not recognize. The software works perfectly in the context it was designed for, and fails in the context it is deployed in.

IshemaLink was designed from Rwanda's actual logistics reality as the starting point, not as an afterthought.

3.2 Mobile Money is the Infrastructure, Not the Alternative

Generic logistics platforms treat mobile money as an optional add-on. In Rwanda, MTN Mobile Money and Airtel Money are not alternatives to bank transfers; they are the primary financial infrastructure for the majority of traders, transport agents, and farmers moving produce between districts.

IshemaLink's entire payment architecture is built around this reality. The BookingService does not confirm a booking until the MoMo payment is verified. The MomoMock gateway simulates the exact push prompt and webhook callback flow used by MTN and Airtel in production. The handle_payment_callback method handles the asynchronous nature of mobile money: the user approves on their phone, the network processes it, and the webhook arrives seconds later without blocking the main application thread.

Critically, failed payments are handled explicitly. The shipment transitions to payment failed, no booking is confirmed, and the driver is not locked out of availability. This prevents the common failure mode of generic platforms where a failed payment leaves the system in an inconsistent state, a driver marked as assigned to a shipment that was never paid for.

3.3 Government Compliance is Embedded, Not Optional

Rwanda's logistics sector operates under active government oversight. RURA regulates vehicle licensing. RRA requires Electronic Billing Machine receipts for every commercial transaction. The EAC requires standardized customs manifests for cross-border shipments.

Generic platforms ignore this entirely, generating PDF invoices that satisfy no regulatory requirement in Rwanda. Businesses using such platforms must maintain a parallel paper-based compliance system alongside their digital operations, doubling administrative work and creating audit exposure.

IshemaLink embeds compliance into the booking workflow. The gov/rura/verify-license/ endpoint checks driver license validity before dispatch. The gov/ebm/sign-receipt/ endpoint requests an RRA digital signature for every completed payment. The gov/customs/generate-manifest/ endpoint produces EAC-compliant XML for international shipments. These are not optional features; they are integrated through the same BookingService that handles payment and driver assignment.

3.4 Infrastructure Must Match Real Conditions

Generic platforms assume continuous GPS tracking with updates every few seconds. This works on Kigali's paved roads with strong 4G. It does not work reliably on the road between Huye and Nyanza, or in the hills around Musanze during the rainy season.

IshemaLink's tracking endpoint is designed for intermittent update cycles. The architecture anticipates that a driver's device syncs location in batches when connectivity is available, rather than assuming a constant stream. The tracking response always includes a timestamp so the recipient knows exactly how recent the data is.

SMS is the primary notification channel, not because email is unavailable, but because SMS works on basic phones without a smartphone, functions on 2G, and delivers even when internet connectivity is poor. Email is used for exporters and business owners with reliable data access. This channel prioritization is built directly into the `NotificationEngine` design and reflects how communication actually works across Rwanda's diverse connectivity landscape.

3.5 Data Sovereignty is a Design Requirement

Rwanda's data sovereignty requirements mean that logistics data, financial records, agent information, cargo manifests, and government compliance logs must remain on infrastructure physically located within Rwanda. For a platform handling real financial transactions and government integrations, this is not a bureaucratic checkbox. It is a fundamental trust condition between the platform and the agencies whose data it processes.

IshemaLink's production stack is designed for AOS Rwanda or KTRN deployment. The MinIO backup system runs within Rwanda's borders. No data pipeline routes through foreign cloud providers for primary storage. This constraint shapes every infrastructure decision in the production stack.

3.6 Conclusion

Generic logistics software fails in Rwanda, not because Rwanda is an edge case, but because those platforms were designed for a minority of the world's logistics reality. The assumptions baked in bank accounts, continuous connectivity, Western regulatory frameworks, and smartphone-first users do not describe most of Rwanda's logistics actors.

IshemaLink succeeds because it treats Rwanda's context as the design specification rather than the deployment constraint. Mobile money is not integrated as a workaround; it is the payment architecture. Government compliance is not added at the end; it is embedded in the core booking flow. Offline resilience is not a fallback; it is the expected operating condition. The result is a platform built for the agents, drivers, exporters, and government officials who actually use it.