

Platform Modernization Assessment

Executive Summary

Following a detailed technical review of the existing PartsHub platform, several architectural limitations have been identified that may impact long-term scalability, maintainability, development velocity, infrastructure costs, and future feature expansion.

The current platform is built on a Python/Django architecture with a large monolithic structure consisting of more than 229 database tables and highly coupled application modules. While functional, the architecture introduces increasing complexity as the platform grows.

A migration to a modern TypeScript-based architecture using NestJS provides significant advantages in modularity, type safety, real-time capabilities, scalability, infrastructure efficiency, and developer productivity.

This document outlines the key architectural observations and the advantages of adopting NestJS as the next-generation backend platform.

1. Monolithic Architecture and Excessive File Bloat

Current Architecture Challenges

The current codebase contains several oversized "God Files" that violate Single Responsibility Principles.

Examples include:

File	Approximate Size
accounts/models.py	3,700+ lines
core/models.py	2,300+ lines
claims/models.py	1,800+ lines

The accounts module alone defines approximately 75 database tables within a single file.

This creates:

- Difficult maintenance
- Higher onboarding time
- Increased merge conflicts
- Reduced code discoverability
- Higher defect rates

NestJS Advantage

NestJS enforces Domain-Driven Design (DDD) principles through:

- Modules
- Controllers
- Services
- Repositories
- Entities

Example structure:

```
claims/  
├── claims.module.ts  
├── claims.controller.ts  
├── claims.service.ts  
├── entities/  
├── dto/  
└── interfaces/
```

Benefits:

- Clear ownership boundaries
- Improved maintainability
- Easier onboarding
- Better testability
- Reduced technical debt

2. Lack of Native End-to-End Type Safety

Current Architecture Challenges

Python is dynamically typed.

Across 229 tables and hundreds of APIs, data mismatches may only surface during runtime.

Common production issues include:

- KeyError

- TypeError
- Payload mismatch
- Missing field exceptions
- Serialization inconsistencies

These errors are often discovered late in QA or production environments.

NestJS Advantage

NestJS leverages TypeScript for strict compile-time validation.

Benefits include:

Shared Types

Frontend and backend share:

- DTOs
- Interfaces
- Validation schemas

Compile-Time Safety

Schema changes immediately surface during build time.

Example:

```
UpdateGarageDto
GarageResponse
SupplierRFQDto
AuctionBidDto
```

Benefits:

- Fewer production bugs
- Faster development
- Safer refactoring
- Strong API contracts

3. Real-Time Auction and Chat Scalability

Current Architecture Challenges

PartsHub includes:

- Salvage Auctions
- Live Bidding
- Chat Systems

- Viewer Counts
- Notifications

Current implementation relies on:

- Django
- Pusher
- HTTP Polling

Challenges:

- Additional licensing costs
- Latency
- Vendor dependency
- Scaling limitations

NestJS Advantage

Node.js is built around a non-blocking event loop.

NestJS provides:

- Socket.io
- WebSocket Gateways
- Redis Pub/Sub
- Redis Streams

Benefits:

- Thousands of concurrent bidders
- Sub-second updates
- Reduced infrastructure costs
- Elimination of Pusher dependency

4. Background Job Processing Limitations

Current Architecture Challenges

Background processing currently relies on APScheduler running within application processes.

Typical jobs include:

- CSV Imports
- Auction Closing
- Notification Delivery
- Scheduled Reports
- License Expiry Monitoring

Potential risks:

- Duplicate execution
- Race conditions
- Memory leaks
- Multi-node inconsistencies

NestJS Advantage

NestJS integrates with:

- BullMQ
- Redis Queues
- Worker Threads

Capabilities:

- Retry mechanisms
- Exponential backoff
- Dead-letter queues
- Job monitoring
- Distributed execution

Benefits:

- Reliable processing
- Horizontal scalability
- Fault tolerance

5. Complex Profile Versioning Architecture

Current Architecture Challenges

The current platform contains over 25 specialized versioning tables.

Examples:

- organization_company_versions
- organization_bank_account_versions
- organization_vat_versions
- organization_working_hours_versions

Simple profile changes trigger:

- Multiple inserts
- Version increments
- Approval chains
- Additional locking

Challenges:

- Increased schema complexity
- Slower queries
- Higher maintenance effort

NestJS Advantage

Modern audit approaches include:

Snapshot-Based Audit Logs

Store historical changes as:

- JSONB snapshots
- Event records
- Temporal history

Benefits:

- Reduced table count
- Simplified schema
- Faster development
- Easier reporting

Potential reduction:

- 40%+ fewer versioning tables

6. Infrastructure Efficiency and Memory Usage

Current Architecture Challenges

Gunicorn-based deployments typically require:

- Multiple worker processes
- Large memory allocations

Estimated consumption:

Workers Approximate RAM

8 2GB+

16 4GB+

Challenges:

- Higher AWS costs
- Increased scaling complexity
- Idle resource consumption

NestJS Advantage

Node.js uses:

- Single event loop
- Non-blocking I/O
- Efficient memory model

Typical consumption:

Instance	Approximate RAM
NestJS Service	80–150 MB

Benefits:

- Lower hosting costs
- Better scaling efficiency
- Improved concurrency

7. Media Delivery Bottlenecks

Current Architecture Challenges

Media access currently passes through backend authorization layers.

Examples:

- `secure_media_view.py`
- `admin_media_resolve_view.py`

Challenges:

- Backend worker consumption
- Increased latency
- Bandwidth costs
- Reduced scalability

NestJS Advantage

Modern cloud architecture uses:

- AWS S3

- CloudFront
- Signed URLs
- Edge Authorization

Benefits:

- Direct media delivery
- Reduced API load
- Lower bandwidth costs
- Global CDN performance

8. Circular Dependencies and Tight Coupling

Current Architecture Challenges

Multiple domains are tightly connected through direct imports and foreign key relationships.

Examples:

- Claims ↔ Garage
- Garage ↔ Orders
- Orders ↔ Suppliers
- Claims ↔ Job Cards

Challenges:

- Circular imports
- Difficult testing
- Complex deployments
- Difficult service extraction

NestJS Advantage

NestJS promotes decoupling through:

- Dependency Injection
- Event Emitters
- Interface Contracts
- Domain Services

Benefits:

- Cleaner architecture
- Better separation of concerns
- Easier future microservice extraction

9. Full-Stack Development Velocity

Current Architecture Challenges

Current technology stack requires:

- Python backend developers
- JavaScript frontend developers

Challenges:

- Larger hiring requirements
- Knowledge silos
- Communication overhead

NestJS Advantage

Single language across the platform:

TypeScript Everywhere

- NestJS Backend
- Next.js Frontend
- Shared DTOs
- Shared Validation

Benefits:

- Faster onboarding
- Smaller teams
- Improved collaboration
- Higher productivity

10. Modern ORM and Integration Ecosystem

Current Architecture Challenges

Django ORM frequently requires optimization using:

- `select_related`
- `prefetch_related`

Risks:

- N+1 queries
- Hidden performance issues
- Difficult query tracing

Additionally:

- Kafka integration
- gRPC integration
- Event streaming

require additional libraries and custom implementations.

NestJS Advantage

Modern ecosystem support includes:

Prisma ORM

- Type-safe queries
- Auto-generated models
- Query transparency

Native Integrations

- REST APIs
- GraphQL
- gRPC
- Kafka
- RabbitMQ
- Event-Driven Architectures

Benefits:

- Faster integrations
- Improved performance
- Better observability

Architectural Comparison Summary

Capability	Current Django Platform	Proposed NestJS Platform
Language	Python 3.12 (Dynamic)	TypeScript 5+ (Static)
Architecture	Large Monolith	Modular Monolith / DDD
Type Safety	Runtime Validation	Compile-Time Validation
Real-Time Features	Pusher + Polling	Native WebSockets
Background Jobs	APScheduler	BullMQ + Redis
Scalability	Multi-Process Scaling	Event-Driven Scaling
Memory Usage	High	Low
Media Delivery	Backend Proxying	CDN + Signed URLs
Team Structure	Backend + Frontend Specialists	Unified TypeScript Team
API Ecosystem	Traditional REST	REST, GraphQL, gRPC, Kafka
Future Microservices	Difficult	Straightforward

Database Architecture Assessment

Executive Summary

A detailed review of the existing PartsHub database architecture reveals significant schema complexity and over-engineering that has resulted in a database containing approximately **229 tables**.

For a platform whose primary business domains consist of:

- Insurers
- Garages
- Suppliers
- Claims
- RFQs
- Quotations
- Orders
- Job Cards
- Inventory
- Auctions
- Payments

an enterprise-grade architecture would typically require approximately **45–55 carefully designed tables**.

The current implementation introduces unnecessary complexity, increased maintenance costs, slower development velocity, and long-term scalability challenges.

This document identifies the primary architectural concerns and proposes a streamlined schema strategy suitable for a modern NestJS + PostgreSQL platform.

Current State Overview

Database Size

Metric	Current Platform
Total Tables	~229
Major Business Domains	~10
Estimated Required Tables	~85–95

Metric	Current Platform
Potential Reduction	55%+

The current schema contains a large number of duplicate, overlapping, and highly specialized tables that increase operational complexity without delivering proportional business value.

Major Architectural Issues

1. Excessive Versioning Tables

Current Approach

The platform maintains separate versioning and counter tables for nearly every profile component.

Examples:

- organization_company_versions
- organization_company_version_counters
- organization_bank_account_versions
- organization_bank_account_version_counters
- organization_vat_versions
- organization_vat_version_counters
- organization_license_versions
- organization_license_version_counters
- organization_working_hours_versions
- organization_working_hours_version_counters

and many more.

Estimated impact:

- 30+ tables dedicated solely to version tracking

Problems

- Increased migration complexity
- Additional joins
- More foreign key dependencies
- Difficult reporting
- Higher maintenance effort

Recommended Architecture

Implement a centralized audit mechanism.

Example:

profile_change_requests
entity_audit_logs
audit_events

Using:

- PostgreSQL JSONB
- Snapshot storage
- Event history

Benefits:

- Single audit framework
- Simplified schema
- Reduced maintenance

Expected reduction:

30+ tables → 1–2 tables

2. Vehicle Mapping Duplication

Current Approach

Separate mapping tables exist for garages and suppliers.

Examples:

Garage mappings:

- garage_vehicle_type_mapping
- garage_vehicle_brand_mapping
- garage_vehicle_model_mapping
- garage_vehicle_variant_mapping

Supplier mappings:

- supplier_vehicle_type_mapping
- supplier_vehicle_brand_mapping
- supplier_vehicle_model_mapping
- supplier_vehicle_variant_mapping

Total:

Approximately 22 tables.

Problems

- Duplicate business logic
- Duplicate queries
- Increased maintenance
- Redundant indexing

Recommended Architecture

Implement a unified capability structure.

Example:

organization_specialties

Columns:

organization_id
vehicle_node_id
specialty_type

Benefits:

- Single source of truth
- Simplified queries
- Easier expansion

Expected reduction:

22 tables → 1–2 tables

3. Duplicate Business Entities

Current Approach

Several domains contain overlapping entities representing the same business concept.

Examples:

Garage

garage.Garage
accounts.Organization

RFQ Assignment

SupplierRFQAssignment
RFQAssignedSupplier

RFQ Management

SupplierRFQ

GarageRFQ

Problems

- Confusing ownership
- Duplicate business rules
- Increased development effort
- Data consistency risks

Recommended Architecture

Unified domain entities.

Example:

rfqs

with:

source_type
initiator_type
target_type

Benefits:

- Cleaner architecture
- Reduced duplication
- Easier reporting

4. Fragmented Rating and Review System

Current Approach

Multiple review systems exist.

Examples:

- orders_garage_order_reviews
- orders_supplier_order_reviews
- garage_insurance_garage_ratings

Problems

- Duplicate rating logic
- Multiple APIs
- Inconsistent reporting

Recommended Architecture

Single reviews table.

Example:

reviews

Structure:

id
target_type
target_id
rating
review_text
created_by

Benefits:

- Unified review engine
- Reusable APIs
- Easier analytics

Expected reduction:

3+ tables → 1 table

5. Over-Engineered Hot Deals Module

Current Approach

A secondary feature currently uses approximately 11 dedicated tables.

Examples:

- insurance_hot_deals
- insurance_hot_deal_items
- insurance_hot_deal_media
- insurance_hot_deal_status_history
- insurance_hot_deal_audit_log
- hot_deal_carts
- hot_deal_cart_items
- insurance_hot_deal_orders
- insurance_hot_deal_order_items

Problems

- Excessive schema footprint
- Increased maintenance burden

- Feature complexity disproportionate to business value

Recommended Architecture

Simplified model:

hot_deals
hot_deal_items
hot_deal_orders

Expected reduction:

11 tables → 3 tables

Operational Impact of 229 Tables

Database Complexity

Typical profile retrieval may require:

- 10–15 joins
- Multiple nested queries
- Additional lookup tables

Result:

- Increased database load
- Slower API response times

Developer Productivity

New developers face challenges understanding:

- Entity ownership
- Table relationships
- Business rule locations

Result:

- Longer onboarding
- Higher development costs
- Increased bug rates

Migration Risk

Large numbers of interconnected tables create:

- Migration locks
- Circular dependencies
- Production deployment risks

Result:

- Increased operational overhead

Infrastructure Cost

More joins and larger object graphs lead to:

- Higher CPU usage
- Increased memory consumption
- Larger database instances

Result:

- Higher cloud costs

AI Architecture Strategy

Executive Summary

A common misconception in software architecture is that if a platform plans to use Artificial Intelligence in the future, the entire application should be built using Python from day one.

Modern enterprise platforms no longer follow this approach.

Industry leaders such as Uber, Airbnb, Stripe, Netflix, Shopify, and many large SaaS companies separate:

- Core Business Systems
- AI/ML Workloads

into independent services with distinct responsibilities.

For PartsHub, the recommended architecture is:

- **NestJS (TypeScript)** for all transactional business operations
- **Python AI Services** only where machine learning or computer vision is required

This approach provides superior reliability, scalability, maintainability, and infrastructure efficiency.

Why AI Should Be a Separate Service

1. AI Failures Should Never Impact Core Operations

Monolithic Approach

If AI processing runs inside the main application:

- High memory usage
- Model crashes
- GPU failures
- Long-running inference jobs

can impact:

- Claims
- Orders
- Payments
- RFQs
- Auctions

A single AI issue can potentially affect the entire platform.

Decoupled Architecture

With NestJS and a dedicated AI service:

- Core platform remains operational
- Orders continue processing
- Auctions remain active
- Users stay online

Only the AI feature is temporarily unavailable.

Business Benefit

Improved reliability and business continuity.

2. Significant Infrastructure Cost Savings

Different Workloads Need Different Hardware

Business Operations

Activities such as:

- CRUD operations
- User management
- RFQs
- Quotations
- Orders
- Payments

require:

- Fast I/O
- Moderate CPU
- Low memory

AI Workloads

Activities such as:

- Image recognition
- OCR
- Machine learning inference

require:

- GPUs
- CUDA drivers
- High-memory instances

Problem with a Single Platform

Combining both workloads forces the entire application onto expensive infrastructure.

Recommended Solution

Use:

- Low-cost NestJS application servers
- On-demand AI infrastructure

Benefits:

- Lower AWS costs
- Better resource utilization
- Easier capacity planning

3. Most Modern AI Is Consumed Through APIs

Current Industry Trend

The majority of AI functionality today is consumed via APIs rather than built from scratch.

Examples include:

Large Language Models

- OpenAI
- Google Gemini
- Anthropic Claude
- AWS Bedrock

OCR Services

- AWS Textract
- Google Document AI
- Azure Document Intelligence

Speech & Translation

- Whisper
- Azure Speech Services
- Google Speech APIs

NestJS Integration

NestJS provides mature SDK support for all major AI providers.

Example:

```
const summary = await openai.chat.completions.create(...)
```

No Python dependency is required for these integrations.

4. Avoiding Dependency Conflicts

The Challenge

AI frameworks introduce large dependency chains.

Examples:

- PyTorch
- TensorFlow
- CUDA
- OpenCV
- NumPy
- Transformers

These often require:

- Specific Python versions
- GPU drivers
- OS-level dependencies

Common Problems

- Deployment failures
- Package conflicts
- Larger Docker images
- Slower CI/CD pipelines

Docker image sizes frequently exceed:

- 5–10 GB

for AI-enabled environments.

Decoupled Architecture Benefits

The business platform remains lightweight.

Typical NestJS deployment:

- Fast builds
- Smaller containers
- Simpler maintenance

AI dependencies remain isolated in dedicated services.

5. Independent Scaling

Real-World Scenario

At 9:00 AM:

- 500 garages submit RFQs
- Hundreds of suppliers respond
- Thousands of API requests arrive

The platform should scale immediately.

NestJS Strength

Node.js excels at:

- Concurrent requests
- Event-driven workflows
- WebSockets
- Real-time systems

AI Workload Example

Only:

- 10 vehicle inspections
- 15 OCR requests

may occur during the same period.

These workloads do not require the entire platform to scale.

Independent Scaling Model

Core Platform:

10 → 20 → 50 Containers

AI Service:

1 → 2 GPU Containers

Each component scales according to demand.

Conclusion

After a comprehensive assessment of the current PartsHub platform, it is recommended that the next phase of the product be built on a modern **NestJS + TypeScript architecture**, supported by a streamlined PostgreSQL database and isolated AI services.

The current platform has evolved into a highly complex monolithic system with approximately **229 database tables**, extensive cross-module dependencies, duplicated domain models, and significant operational overhead. While the platform remains functional, the existing architecture introduces unnecessary complexity that will continue to increase development costs, reduce agility, and limit long-term scalability.

A redesigned architecture based on **NestJS, TypeScript, PostgreSQL, Redis, and modular domain boundaries** will provide a significantly stronger foundation for future growth, enterprise adoption, and large-scale deployment.

Database Architecture Simplification

The current database architecture contains approximately **229 tables**, significantly exceeding what is typically required for a platform of this scope.

A review of the schema indicates several opportunities for consolidation, including:

- Excessive versioning tables
- Duplicate mapping structures
- Redundant domain entities
- Fragmented review systems
- Over-engineered secondary modules
- Tight relational coupling across business domains

By redesigning the platform around clean domain boundaries and modern PostgreSQL capabilities, the schema can be streamlined to approximately **45–50 well-structured tables** without sacrificing functionality.

This simplification will result in:

- Improved maintainability
- Better query performance
- Reduced database complexity
- Faster onboarding for new developers
- Lower infrastructure overhead
- Simplified deployment and migration processes

Most importantly, it creates a database structure that can evolve naturally as the platform grows.

Recommended Technology Architecture

Core Platform

The core business platform should be implemented using:

Backend

- NestJS
- TypeScript
- Prisma ORM
- PostgreSQL

Real-Time & Background Processing

- Redis
- BullMQ
- Socket.io

Frontend

- Next.js
- TypeScript
- Tailwind CSS

Infrastructure

- AWS ECS / Kubernetes
- CloudFront
- S3
- Redis Cluster

This architecture supports high-performance transactional workloads while maintaining clean separation between business domains.

AI Strategy and Technology Direction

The presence of future Artificial Intelligence requirements should **not** dictate the technology choice for the entire platform.

Modern enterprise systems separate:

- Core Business Operations
- AI and Machine Learning Workloads

into independent services.

The recommended approach is:

Core Platform

NestJS + TypeScript

Responsible for:

- Authentication
- Organizations
- Claims
- RFQs
- Quotations
- Orders
- Inventory
- Job Cards
- Auctions
- Payments
- Notifications
- Real-Time Communication

AI Layer

Python-Based AI Microservices

Responsible for:

- Vehicle Damage Detection
- OCR Processing
- Document Extraction
- Predictive Analytics
- Machine Learning Models
- Fraud Detection
- Future AI Initiatives

External AI Providers

Where appropriate, leverage:

- OpenAI
- Google Gemini
- Anthropic Claude
- AWS Bedrock
- AWS Textract

through secure API integrations rather than embedding AI directly into the core platform.

Benefits of the Proposed Architecture

The combination of **NestJS for transactional business operations** and **isolated Python AI services for machine learning workloads** delivers significant advantages across multiple dimensions.

Higher Reliability

AI failures or resource-intensive workloads cannot impact critical business functions such as claims processing, RFQs, orders, auctions, or payments.

Lower Infrastructure Costs

Business workloads run on lightweight application servers while AI workloads can scale independently based on demand.

Faster Development

A unified TypeScript ecosystem accelerates feature delivery, onboarding, testing, and long-term maintenance.

Better Scalability

Core platform services and AI services can scale independently, allowing infrastructure to grow according to actual workload requirements.

Easier Maintenance

Clear separation of concerns reduces technical debt and simplifies ongoing development.

Future-Proof AI Integration

New AI capabilities can be introduced as standalone services without affecting the stability of the core platform.

Strategic Outcome

The proposed architecture provides a clear path toward:

- Enterprise-grade scalability
- Reduced operational complexity
- Improved developer productivity
- Lower total cost of ownership
- Faster feature delivery
- Greater platform resilience
- Long-term maintainability.