

Comprehensive Production-Level Backend Architecture & Security Code Audit

Target System: PartsHub Automotive API (`core-api-partshub/broker_platform`)

Stack: Python 3.12, Django 5.2.11, Django REST Framework 3.16.1, PostgreSQL, SimpleJWT, APScheduler, Boto3 / AWS S3

Audit Scope: Complete codebase audit covering Architecture, Security, Database, Performance, Scalability, Multi-Tenancy, Error Handling, Authentication/Authorization, Validation, Integrations, Logging, Configuration, Testing, and Deployment Readiness.

Executive Summary

An exhaustive, line-by-line inspection of the PartsHub backend repository was conducted across all applications (`accounts`, `admin_login`, `auctions`, `claims`, `core`, `garage`, `insurance`, `inventory`, `invoices`, `job_cards`, `orders`, `organization_admin`, `payments`, `reviews`, `supplier`, `verification`).

While the codebase exhibits substantial business domain modeling (auctions, claim lifecycles, RFQs, inventory lots, and verification workflows), the implementation suffers from **critical architectural, security, and scalability deficiencies** that make it **unfit for enterprise production deployment in its current state**.

Key Audit Findings:

1. Critical Security Vulnerabilities (Blockers):

- **Cross-Tenant IDOR Bypass:** Any user belonging to an organization marked as "HQ" completely bypasses tenant isolation in claims endpoints (`ClaimDetailByIdView`, `ClaimFullSummaryView`, `ClaimCloseView`, `ClaimTimelineView`) and can read or mutate claims belonging to any organization across the platform.
- **Password Reset Account Takeover:** The `ForgotPasswordView` returns the generated plain OTP directly inside the HTTP JSON response payload (`"OTP": str(otp_token.otp_plain)`), allowing complete account takeover for any user.
- **Plaintext OTP Storage & Leakage:** OTPs are stored in plaintext in the database table `OTPAccessToken` (`hash_otp` is defined but never called) and printed to standard output via raw `print` statements.
- **Super Admin Token Lifetime:** Admin JWT access tokens are hardcoded with a **30-day lifespan** (`access.set_exp(lifetime=timedelta(days=30))`).
- **Unauthenticated Default Endpoints:** DRF's `DEFAULT_PERMISSION_CLASSES` is omitted in `settings.py`, defaulting to `AllowAny`. Views without explicit `permission_classes` are accessible to unauthenticated callers.

- **Insecure Production Defaults:** Permissive wildcard defaults for `CORS_ALLOW_ALL_ORIGINS = True`, `ALLOWED_HOSTS = ["*"]`, and `PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED = True`.

2. Severe Scalability & Concurrency Bottlenecks:

- **Database Row Locks Held During Synchronous External HTTP Calls:** Auction bidding locks rows using `SELECT FOR UPDATE` within `@transaction.atomic` and invokes synchronous HTTP Pusher triggers before committing, blocking all competing bidders on the auction.
- **In-Memory Data Processing:** Core endpoints (e.g. `InsuranceClaimOrderListView`) load entire database result sets, serialize every object in memory, and perform in-memory substring filtering, sorting, and pagination in Python rather than utilizing PostgreSQL query operators.
- **In-Process Multi-Worker Background Scheduler:** `APScheduler` runs inside the Gunicorn web worker processes (`threading.Timer(1.0, start_scheduler).start()`), resulting in competing scheduler instances per worker thread, race conditions on `DjangoJobStore`, and abrupt termination of background imports upon worker recycling.
- **Synchronous External Blocking Calls:** Twilio SMS, Pusher notifications, and SMTP email transmissions run synchronously inside request handlers without timeouts, retry mechanisms, or asynchronous task queues (no Celery/RabbitMQ/Redis).

3. Structural & Architectural Debt:

- Massive God models (e.g., `accounts/models.py` at 3,763 lines / 126 KB).
- Inconsistent soft-delete abstraction (`TimeStampedSoftDeleteModel` defines zero soft-delete fields).
- Near-zero automated test coverage across authentication, authorization, and core transactional logic.
- Dual un-unified user identity models (`OrganizationUser` vs `django.contrib.auth.models.User`).

Systematic Category Analysis

1. Architecture

- **Layer Separation:** Layering is severely inconsistent. While some modules attempt a routes → services → models structure, business logic is heavily fragmented. Controllers (views) perform direct ORM queries, construct complex query dictionaries, and execute state transitions inline.

- **God Modules & God Classes:** `accounts/models.py` spans 3,763 lines defining dozens of disparate entities (users, organizations, KYC, versioning counters, tokens, addresses, banks). View files such as `inventory/views/hot_deal_views.py` (50 KB) and `claims/views/rfq_step_views.py` (30 KB) act as monolithic multi-action controllers.
- **Tight Coupling:** Views depend directly on specific model field names and internal query shapes rather than repository contracts or domain service abstractions.

2. API Design

- **REST Conventions:** Endpoints mix RESTful resource hierarchies with RPC-style action routes (e.g., `/api/v1/auth/forgot-password/`, `/api/v1/claims/<id>/close/`).
- **Response Consistency:** Response payloads use varying envelope keys (`detail`, `data`, `results`). Pagination envelopes are manually stitched in views (`paginate_list` vs `paginate_queryset` vs DRF standard pagination).
- **Error Consistency:** Settings claim `{success: false, code, error_message, detail?}`, but DRF default validation errors bypass this structure, returning raw dictionary maps `{field: ["error"]}`.

3. Security

- **Broken Object-Level Authorization (IDOR):** The tenant scoping rule `if hq_or_branch != "HQ": filters["organization_id"] = caller_org_id` fails to bind an HQ user to their specific organization hierarchy, exposing all tenant records to any HQ user.
- **Credential & Token Exposure:** Plaintext OTP generation and response exposure in `password_reset_views.py`; 30-day admin access tokens; plaintext OTP logging to standard output.
- **Input & File Security:** Absence of file magic-byte inspection allows arbitrary executable files disguised as `.png` or `.pdf` to be stored and served via presigned/proxy URLs.
- **Container Security:** The production Dockerfile creates an unprivileged user `appuser` but explicitly executes the process as `root`.

4. Database

- **Transaction Safety & Row Locking:** Long-running external network I/O executed inside `@transaction.atomic` blocks.
- **ORM & Query Efficiency:** Multiple N+1 queries in serializers (e.g., `verification/serializers/checklist_serializers.py:56` running `OrganizationDocuments.objects.filter(...)` per serialized item).
- **Vendor Lock-in via Raw SQL:** Database sequence numbering (`SELECT`

`nextval(...)` directly relies on PostgreSQL-specific sequence objects via `cursor.execute()`.

- **Soft Deletion Flaw:** Base model `TimeStampedSoftDeleteModel` defines only timestamps, lacking `is_deleted` or `deleted_at` fields, leading to inconsistent application of soft delete logic.

5. Error Handling

- **Swallowed Exceptions:** Services frequently catch generic `Exception` and return boolean tuples `(False, str(e))` or print to stdout without recording stack traces.
- **Handler Leaks:** Default DRF handler returns `None` for unhandled exceptions, rendering Django HTML 500 error pages in production WSGI environments. In `admin_auth_views.py`, internal exception strings (`detail={"error": str(e)}`) are returned to the client on 500 errors.

6. Performance

- **Synchronous Blocking Operations:** Third-party integrations (Twilio REST API, Pusher WebSocket triggers, SMTP email) execute synchronously within the HTTP thread.
- **Memory Consumption:** Endpoints perform complete queryset retrieval and in-memory list sorting/filtering (`orders/views/insurance_claim_order_list_view.py`).
- **Middleware Overhead:** `AuditTrailMiddleware` performs duplicate JWT authentication queries and synchronous DB snapshots on every mutating request.

7. Scalability

- **Process Model Bottlenecks:** Gunicorn is configured with 3 workers and a 120-second timeout. With blocking network calls and in-memory query processing, fewer than 5 concurrent slow requests will exhaust the entire server's processing capacity.
- **Background Workload Coupling:** `APScheduler` runs within the WSGI server process rather than as an independent distributed worker pool (e.g., Celery/Redis).

8. Multi-Tenancy

- **Tenant Leakage:** High-severity multi-tenant data leakage identified across claims endpoints where HQ users bypass tenant filtering completely.
- **Lack of Global Tenancy Enforcement:** The application does not utilize tenant-aware database managers or middleware-enforced tenant query filtering.

9. Authentication & Authorization

- **Dual User Identity:** The system maintains two disjoint authentication systems: `django.contrib.auth.models.User` (used by `admin_login` and Django Admin) and `accounts.models.OrganizationUser` (used by organization users).
- **Incompatible Authenticators:** `DEFAULT_AUTHENTICATION_CLASSES` only specifies

OrganizationJWTAuthentication, causing admin tokens to fail across general endpoints.

- **Missing Object Permissions:** DRF's `has_object_permission` hook is implemented only in the `job_cards` module, while all other apps rely on error-prone manual filter queries in view functions.

10. Validation

- **MIME Type Spoofing:** File validation checks client-reported HTTP headers and file extensions rather than file contents (magic numbers).
- **OTP Validation:** Serializers enforce `max_length=6` on OTP strings without `min_length=6`, accepting empty or truncated tokens.

11. External API Integrations

- **Missing Timeouts & Resilience:** Neither Twilio nor Pusher client initializations define socket timeouts, exponential backoff, or circuit breakers.
- **Hardcoded Configuration:** Twilio SMS service prepends hardcoded +91 country code, breaking SMS delivery for international and UAE numbers.

12. Logging & Observability

- **Unstructured Logging:** Raw `print` statements used across production paths (e.g., `pusher.py`, `otp_utils.py`).
- **Missing Distributed Tracing:** No request correlation IDs (`X-Correlation-ID`) or distributed tracing context to connect HTTP requests to background scheduler tasks and external API calls.

13. Configuration

- **Unsafe Fallbacks:** Wildcard defaults for `ALLOWED_HOSTS = ["*"]` and `CORS_ALLOW_ALL_ORIGINS = True`.
- **Payment Gateway Stubs:** Production settings default to allowing unsigned webhooks from the stub gateway (`PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED = True`).

14. Code Quality

- **Dead Code:** Unused security functions (e.g., `hash_otp` in `otp_utils.py`).
- **Monolithic File Size:** Files exceeding thousands of lines (`accounts/models.py`: 3,763 lines).
- **Mutable Request Manipulation:** Views mutate `request.data` directly.

15. Testing

- **Critical Test Absence:** Near-zero automated test coverage. The core `accounts` module

contains only a 56-line test verifying a fallback vehicle mapping function, with zero tests for authentication, authorization, or multi-tenancy.

16. Deployment & Production Readiness

- **Startup Race Conditions:** Web worker containers execute database migrations on container launch (`python manage.py migrate`), risking deadlock in multi-replica deployments.
 - **Privilege Escalation Risk:** Docker containers run as the root user.
-

Top 20 Issues

Issue 01 — Critical Cross-Tenant Authorization Bypass (BOLA / IDOR) via Broken HQ Scope Logic

- **Category:** Multi-Tenancy & Authorization
- **Severity:** Critical
- **File:** `claims/views/claim_detail_view.py` (and duplicated in `claim_full_summary_view.py`, `claim_close_view.py`, `claim_timeline_view.py`)
- **Function/Class:** `ClaimDetailByIdView.get`
- **Line:** 51–62
- **Problem:** When an authenticated user's organization has `hq_or_branch == "HQ"`, the view omits the `organization_id` filter completely, allowing any HQ user to access claims belonging to any tenant in the database.
- **Evidence:**

```
hq_or_branch = request.auth.get("hq_or_branch", "")
caller_org_id = request.auth.get("org_id")

filters = {"id": claim_id, "is_deleted": False}
if hq_or_branch != "HQ":
    filters["organization_id"] = caller_org_id

claim = Claim.objects.select_related("organization").filter(**filters).first()
```

- **Why this is a problem:** In a multi-tenant platform with multiple independent insurance companies, garages, and suppliers, every tenant can have an HQ organization. When `hq_or_branch == "HQ"`, `filters["organization_id"]` is never populated. The database query simply executes `filter(id=claim_id, is_deleted=False)`. It does not restrict access to the caller's organization or its branches.
- **Production impact:** Complete cross-tenant data breach. Any user from Insurance Company A (or any Garage/Supplier HQ) can view policyholder details, driver info, and repair costs

of Insurance Company B. In `ClaimCloseView`, an HQ user from one company can prematurely close claims belonging to competing insurance companies.

- **Recommended fix:** Use the centralized tenant scope utility `accounts.utils.org_scope.expand_org_scope(request.user)`. Always filter queries with `organization_id__in=expand_org_scope(request.user):`

```
allowed_org_ids = expand_org_scope(request.user)
claim = Claim.objects.filter(id=claim_id, is_deleted=False,
organization_id__in=allowed_org_ids).first()
```

- **Priority:** BLOCKER
-

Issue 02 — Account Takeover via Cleartext OTP Return in Password Reset Response

- **Category:** Authentication & API Security
- **Severity:** Critical
- **File:** `accounts/views/password_reset_views.py`
- **Function/Class:** `ForgotPasswordView.post`
- **Line:** 83–90
- **Problem:** The API response for requesting a password-reset OTP includes the plain-text OTP code directly inside the HTTP response body.
- **Evidence:**

```
return success_response(
    message="OTP sent successfully.",
    code="OTP_SENT",
    detail={
        "verification_session_id": str(otp_token.verification_session_id),
        "OTP": str(otp_token.otp_plain),
    },
)
```

- **Why this is a problem:** Password reset OTPs are meant to serve as an out-of-band proof of possession (via email or SMS). Returning the secret OTP directly to the HTTP client that requested the reset defeats the entire security boundary.
- **Production impact:** Critical 1-click Account Takeover. Any attacker knowing a victim's email or mobile number can call `POST /api/v1/auth/forgot-password/`, read the OTP from the JSON response, verify the OTP, and set a new password, compromising any account instantly.
- **Recommended fix:** Remove "OTP" from the response payload immediately. Return only the unguessable session identifier:

```
return success_response(
```

```
message="OTP sent successfully.",
code="OTP_SENT",
detail={"verification_session_id": str(otp_token.verification_session_id)},
)
```

- **Priority:** BLOCKER
-

Issue 03 — Plaintext OTP Database Storage and Sensitive Data Logging

- **Category:** Security / Sensitive Data Exposure
- **Severity:** High
- **File:** accounts/utils/otp_utils.py
- **Function/Class:** handle_otp_generation_or_resend & send_otp
- **Line:** 26, 93, 192
- **Problem:** OTP tokens are stored in the database in plaintext without cryptographic hashing, and are dumped to standard output using raw print statements.
- **Evidence:**

```
# Line 26: defined but never invoked in storage logic
def hash_otp(otp):
    return hashlib.sha256(otp.encode()).hexdigest()

# Line 93:
if not settings.OTP_SEND:
    print(f"{verification_type.capitalize()} OTP: {otp_plain}")

# Line 192:
create_kwargs = {
    ...
    "otp": otp_plain,
}
otp_obj = OTPAccessToken.objects.create(**create_kwargs)
```

- **Why this is a problem:** OTPs are short-lived authentication credentials. Storing them in plaintext violates basic cryptographic standards. If the database is compromised via read-only SQL injection, backup leakage, or insider access, all valid OTPs are instantly usable. Furthermore, printing plaintext secrets to stdout pollutes container logs.
- **Production impact:** Credential leakage across logging pipelines (CloudWatch, Datadog) and database dumps, leading to secondary authentication bypass.
- **Recommended fix:** Store salted cryptographic hashes (e.g. SHA-256 or bcrypt) of the OTP in the database. When verifying, hash the incoming candidate OTP and compare using constant-time comparison `secrets.compare_digest()`. Remove all print statements.
- **Priority:** BLOCKER

Issue 04 — Password Reset Session Replay and Missing Token Invalidation for Admin Accounts

- **Category:** Authentication Lifecycle & Security
- **Severity:** High
- **File:** admin_login/views/admin_reset_password_views.py
- **Function/Class:** AdminResetPasswordView.post
- **Line:** 203–209, 227–234
- **Problem:** The password reset endpoint does not check if the OTP session is active (`is_active=True`), allowing session reuse. Furthermore, resetting the password does not invalidate existing admin JWT tokens.
- **Evidence:**

```
otp_obj = OTPAccessToken.objects.select_for_update().get(
    verification_session_id=verification_session_id,
    is_verified=True,
    otp_type='FORGOT_PASSWORD' # is_active=True is omitted!
)
...
user.set_password(new_password)
user.save()
# Missing: increment_token_version(user)
otp_obj.is_active = False
otp_obj.save()
```

- **Why this is a problem:** Because `is_active=True` is not checked in the query, setting `otp_obj.is_active = False` has no effect on subsequent requests; the same `verification_session_id` can be reused until `expires_at` lapses. Additionally, omitting `increment_token_version(user)` leaves previously issued JWT tokens valid even after a password reset.
- **Production impact:** Replay attacks on password reset sessions. If an admin account was compromised, changing the password fails to revoke the attacker's active JWT tokens.
- **Recommended fix:** Include `is_active=True` in the query and invoke `increment_token_version(user)`:

```
otp_obj = OTPAccessToken.objects.select_for_update().get(
    verification_session_id=verification_session_id,
    is_verified=True,
    is_active=True,
    otp_type='FORGOT_PASSWORD'
)
user.set_password(new_password)
user.save()
increment_token_version(user)
otp_obj.is_active = False
otp_obj.save(update_fields=['is_active'])
```

- **Priority:** BLOCKER
-

Issue 05 — Unrestricted 30-Day Lifespan on Admin JWT Access Tokens

- **Category:** Authentication Security / Token Architecture
- **Severity:** High
- **File:** `admin_login/views/admin_auth_views.py`
- **Function/Class:** `AdminLoginView.post`
- **Line:** 85–86
- **Problem:** Admin JWT access tokens are explicitly granted an extraordinarily long 30-day lifetime upon login.
- **Evidence:**

```
access = refresh.access_token
access.set_exp(lifetime=timedelta(days=30))
```

- **Why this is a problem:** In standard JWT architectures, access tokens are bearer tokens that cannot be revoked without querying a stateful blacklist or version check on every request. Access tokens should be short-lived (15–60 minutes), relying on refresh tokens for renewal. Granting 30 days of lifetime to the most privileged account role in the system (Super Admin) maximizes the window of vulnerability.
- **Production impact:** If an admin access token is captured through browser history, proxy logs, cross-site scripting (XSS), or insider exfiltration, the token remains valid for 30 days without requiring re-authentication.
- **Recommended fix:** Remove the custom 30-day expiration override. Rely on standard, short-lived access tokens (e.g. 15 to 30 minutes) and require refresh token rotation:

```
access = refresh.access_token
access.set_exp(lifetime=timedelta(minutes=30))
```

- **Priority:** HIGH
-

Issue 06 — Global Default `allowAny` Permissions Across DRF Endpoints

- **Category:** Security / Authorization Architecture
- **Severity:** High
- **File:** `broker_platform/settings.py`
- **Function/Class:** `REST_FRAMEWORK` configuration dictionary

- **Line:** 196–212
- **Problem:** `DEFAULT_PERMISSION_CLASSES` is omitted from `REST_FRAMEWORK` configuration, causing DRF to default to `rest_framework.permissions.AllowAny`.
- **Evidence:**

```
REST_FRAMEWORK = {
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
    "EXCEPTION_HANDLER": "core.exception_handler.custom_exception_handler",
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "accounts.authentication.OrganizationJWTAuthentication",
    ),
    "DEFAULT_FILTER_BACKENDS": [
        "django_filters.rest_framework.DjangoFilterBackend",
    ],
    "DEFAULT_THROTTLE_CLASSES": [
        "rest_framework.throttling.ScopedRateThrottle",
    ],
    "DEFAULT_THROTTLE_RATES": {
        "bulk_import": "20/hour",
    },
}
```

- **Why this is a problem:** Security by default dictates that endpoints should be authenticated unless explicitly marked public. Omitting `DEFAULT_PERMISSION_CLASSES` means any endpoint that inadvertently omits `permission_classes` (such as `QuoteListView` in `claims/views/default_quote.py` and `DamageLevelListView` in `claims/views/level_of_damage_view.py`) is completely public to the internet.
- **Production impact:** Accidental exposure of internal endpoints, data enumeration by unauthenticated attackers, and vulnerability to automated scrapers.
- **Recommended fix:** Explicitly enforce `IsAuthenticated` as the default permission class across the entire framework:

```
REST_FRAMEWORK = {
    ...
    "DEFAULT_PERMISSION_CLASSES": (
        "rest_framework.permissions.IsAuthenticated",
    ),
}
```

Public endpoints (login, registration, webhooks) must explicitly declare `permission_classes = [AllowAny]`.

- **Priority:** HIGH

Issue 07 — Permissive Wildcard Defaults for CORS, Allowed Hosts, and Payment Webhooks

- **Category:** Security / Configuration Management

- **Severity:** High
- **File:** `broker_platform/settings.py`
- **Function/Class:** Global configuration
- **Line:** 35, 475, 641–643
- **Problem:** Crucial production security settings default to insecure wildcard values if their environment variables are omitted or misconfigured.
- **Evidence:**

```
# Line 35:
ALLOWED_HOSTS = env.list("ALLOWED_HOSTS", default=["*"])

# Line 475:
CORS_ALLOW_ALL_ORIGINS = env.bool("CORS_ALLOW_ALL_ORIGINS", default=True)

# Line 641:
PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED = env.bool(
    "PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED", default=True,
)
```

- **Why this is a problem:** If an environment variable is omitted in production, Django will run with `ALLOWED_HOSTS=["*"]` (vulnerable to HTTP Host header attacks, cache poisoning, and password reset poisoning), `CORS_ALLOW_ALL_ORIGINS=True` (allowing malicious third-party websites to make cross-origin authenticated requests from victim browsers), and `PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED=True` (allowing arbitrary callers to forge payment success webhooks).
- **Production impact:** Cross-origin data theft, cache poisoning, and fraudulent order completion via spoofed payment webhooks.
- **Recommended fix:** Enforce strict, fail-safe defaults in production settings:

```
ALLOWED_HOSTS = env.list("ALLOWED_HOSTS", default=[])
CORS_ALLOW_ALL_ORIGINS = env.bool("CORS_ALLOW_ALL_ORIGINS", default=False)
PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED =
env.bool("PAYMENTS_WEBHOOK_ALLOW_STUB_UNSIGNED", default=False)
```

- **Priority:** HIGH

Issue 08 — In-Memory Serialization, Filtering, and Pagination of Entire Tables

- **Category:** Performance & Scalability
- **Severity:** High
- **File:** `orders/views/insurance_claim_order_list_view.py`
- **Function/Class:** `InsuranceClaimOrderListView.get, _serialize_orders, _search_orders`

- **Line:** 32–60, 114–125, 156
- **Problem:** Database rows are queried in full, instantiated into DRF serializers in Python memory, filtered via substring search in Python, sorted in Python, and sliced via `paginate_list`.

- **Evidence:**

```
garage_orders = GarageOrder.objects.filter(**garage_filter).select_related(...)
supplier_orders =
SupplierOrder.objects.filter(**supplier_filter).select_related(...)

combined = _serialize_orders(garage_orders, supplier_orders, request)
combined.sort(key=lambda x: x.get("order_date") or "", reverse=reverse)

if search:
    combined = _search_orders(combined, search)

paginated_orders = paginate_list(combined, params, request)
```

- **Why this is a problem:** This bypasses the database index engine entirely. As the database grows to thousands of orders per claim or organization, the web server retrieves every record over the network wire, constructs thousands of dictionary objects in RAM, performs $O(N)$ string searches, and then discards 95% of them to return page 1.
- **Production impact:** Massive RAM consumption, high CPU load, and severe latency spikes. Under concurrent enterprise user access, Unicorn workers will exhaust RAM and terminate with Out-Of-Memory (OOM) errors or 504 Gateway Timeouts.
- **Recommended fix:** Push filtering, searching, sorting, and pagination down to the database using Django ORM QuerySet operations (`Q` objects, `order_by`, and `.count()`), slicing the QuerySet before passing only the current page of objects to the serializer.
- **Priority:** HIGH

Issue 09 — Blocking Synchronous External Network I/O Inside Atomic Database Transactions

- **Category:** Database Concurrency & Reliability
- **Severity:** High
- **File:** `auctions/services/auction_bid_service.py` & `auctions/services/notification_hooks.py`
- **Function/Class:** `AuctionBidService.place_bid`
- **Line:** 26–30, 122–126
- **Problem:** Inside an atomic database transaction holding a pessimistic row lock (`SELECT FOR UPDATE`), the service executes synchronous blocking external HTTP calls to Pusher.
- **Evidence:**

```

@staticmethod
@transaction.atomic
def place_bid(*, auction_id, bidder_user, amount: Decimal, request=None) -> AuctionBid:
    auction = Auction.objects.select_for_update().get(pk=auction_id, is_deleted=False)
    ...
    try:
        from auctions.services.notification_hooks import notify_outbid, notify_new_bid
        notify_outbid(auction=auction, bid=bid)
        notify_new_bid(auction=auction, bid=bid)
    except Exception:
        logger.exception("Auction notification fan-out failed")

```

- **Why this is a problem:** The pessimistic lock on `Auction` prevents any other bidder from placing a bid. If Pusher's API experiences latency (e.g. 500ms to 3s), the database lock is held for the duration of that external HTTP round-trip. All concurrent bids on that live auction are queued and blocked. Furthermore, if the transaction later rolls back, the notifications have already been delivered to clients.
- **Production impact:** Critical lock contention during the final seconds of live auctions (sniping), leading to query timeouts, transaction deadlocks, and missed bids.
- **Recommended fix:** Defer external notification dispatches until AFTER the transaction has committed successfully using Django's `transaction.on_commit()`:

```

transaction.on_commit(lambda: notify_outbid(auction=auction, bid=bid))
transaction.on_commit(lambda: notify_new_bid(auction=auction, bid=bid))

```

- **Priority:** HIGH

Issue 10 — In-Process APScheduler Multi-Worker Race Conditions Inside Web Workers

- **Category:** Architecture, Scalability & Reliability
- **Severity:** High
- **File:** `core/apps.py` & `core/scheduler.py`
- **Function/Class:** `CoreConfig.ready/start_scheduler`
- **Line:** 52–68 (`core/apps.py`), 22–32 (`core/scheduler.py`)
- **Problem:** The background job scheduler is started inside the WSGI application process using an in-process Python thread (`threading.Timer(1.0, start_scheduler).start()`).
- **Evidence:**

```

# core/apps.py
if should_start:
    import threading
    from core.scheduler import start_scheduler

```

```
threading.Timer(1.0, start_scheduler).start()
```

- **Why this is a problem:** In production, Gunicorn runs multiple worker processes (configured for 3 workers in the Dockerfile). Each worker imports the Django app config and starts its own independent APScheduler instance against the shared PostgreSQL DjangoJobStore. This causes multiple worker processes to compete for job locks, execute duplicate background tasks, and cause database deadlock warnings. Furthermore, when Gunicorn recycles a worker due to timeout or memory limits, background jobs (like 5,000-row bulk imports) are abruptly killed mid-execution.
 - **Production impact:** Duplicate notifications, duplicate automated billing/reconciliation runs, failed bulk imports, and unpredictable job execution states.
 - **Recommended fix:** Remove scheduler initialization from `AppConfig.ready()`. Transition background task execution to an enterprise task queue architecture (e.g., Celery or RQ backed by Redis), or run APScheduler strictly as a standalone, single-instance CLI management command container (`python manage.py run_scheduler`).
 - **Priority:** HIGH
-

Issue 11 — Synchronous Blocking Third-Party Integrations in HTTP Lifecycle

- **Category:** External Integrations & Reliability
- **Severity:** High
- **File:** `core/services/sms_services.py`, `core/providers/pusher.py`, `core/services/email_services.py`
- **Function/Class:** `SmsService.send_sms`, `PusherProvider.send`, `EmailService.send_email`
- **Line:** 17–21 (`sms_services.py`), 18 (`pusher.py`), 177 (`email_services.py`)
- **Problem:** External network communications to Twilio, Pusher, and SMTP mail servers are executed synchronously in the HTTP request thread with no timeouts or circuit breakers.
- **Evidence:**

```
# core/services/sms_services.py
client = Client(account_sid, auth_token)
message = client.messages.create(body=template,
    from_=settings.TWILIO_PHONE_NUMBER, to=f"+91{mobile}")

# core/providers/pusher.py
self.client.trigger(channels=channel, event_name=event, data=payload)

# core/services/email_services.py
msg.send()
```

- **Why this is a problem:** Synchronous network calls in HTTP views make the backend's

availability dependent on third-party uptime. If Twilio's API delays for 10 seconds or the SMTP server hangs, the Gunicorn worker thread remains blocked. With only 3 Gunicorn workers, just 3 concurrent requests calling SMS or email can cause 100% downtime for all incoming traffic.

- **Production impact:** Cascading system failure and 504 Gateway Timeouts whenever external vendors experience minor latency or network jitter.
 - **Recommended fix:** Offload all email, SMS, and webhook pushes to background worker queues. If a call must remain synchronous, enforce strict connect and read timeouts (e.g., `timeout=3.0`) and catch network exceptions gracefully.
 - **Priority:** HIGH
-

Issue 12 — Ineffective Custom Exception Handler Leaking HTML and Breaking API Contracts

- **Category:** Error Handling & API Contracts
- **Severity:** High
- **File:** `core/exception_handler.py` & `broker_platform/settings.py`
- **Function/Class:** `custom_exception_handler`
- **Line:** 12–29 (`core/exception_handler.py`), 253–255 (`broker_platform/settings.py`)
- **Problem:** The custom exception handler fails to implement the advertised standard error envelope (`{success: false, code, error_message, detail?}`) and returns `None` on unhandled 500 errors, causing Django to output raw HTML error pages.
- **Evidence:**

```
def custom_exception_handler(exc, context):
    if isinstance(exc, SuspiciousOperation):
        ...
        exc = ParseError(detail=str(exc) or "Bad request.")
    return exception_handler(exc, context)
```

- **Why this is a problem:** DRF's standard `exception_handler` only handles DRF `APIException` subclasses and returns `None` for standard Python errors (`KeyError`, `AttributeError`, database disconnections). When `None` is returned, Django renders standard HTML 500 pages. In addition, DRF's default response shape (`{ "field": ["error"] }`) does not conform to the documented `{success, code, error_message}` schema.
- **Production impact:** Mobile apps and SPA frontends encounter JSON parse errors on backend failures because they receive HTML error pages instead of JSON, preventing graceful user error messaging.

- **Recommended fix:** Rewrite `custom_exception_handler` to catch all unhandled exceptions, log the full traceback with a unique `error_id`, and always return a consistent JSON response:

```
def custom_exception_handler(exc, context):
    response = exception_handler(exc, context)
    if response is None:
        error_id = uuid.uuid4()
        logger.exception("Unhandled Server Error [ID: %s]", error_id)
        return Response({
            "success": False,
            "code": "INTERNAL_SERVER_ERROR",
            "error_message": "An unexpected error occurred.",
            "error_id": str(error_id)
        }, status=500)
    ...
```

- **Priority:** HIGH
-

Issue 13 — Superficial File Upload Validation Bypassing MIME Checking (No Magic Byte Inspection)

- **Category:** Security / File Uploads
- **Severity:** High
- **File:** `core/utils/file_validate.py`
- **Function/Class:** `validate_uploaded_document`
- **Line:** 87–93
- **Problem:** Uploaded file validation checks only the client-submitted HTTP Content-Type header and the filename extension, with zero inspection of the file's binary magic bytes.
- **Evidence:**

```
content_type = getattr(file, "content_type", None)
guessed_type, _ = mimetypes.guess_type(file.name)
if (
    content_type not in allowed_content_types
    and guessed_type not in allowed_content_types
):
    raise ValidationError("Invalid file content type.")
```

- **Why this is a problem:** Both `file.content_type` and `file.name` are completely under the control of the attacker. An attacker can upload an executable file, a PHP script, or a malicious SVG/HTML containing JavaScript simply by naming it `avatar.png` and setting Content-Type: `image/png`.
- **Production impact:** Stored Cross-Site Scripting (XSS) when files are served from S3, potential remote code execution, and distribution of malicious payloads through signed platform links.

- **Recommended fix:** Inspect the actual binary header of the uploaded file using `python-magic` or `filetype` to verify that the file contents match the claimed MIME type before accepting it:

```
import filetype
kind = filetype.guess(file.read(2048))
file.seek(0)
if kind is None or kind.mime not in allowed_content_types:
    raise ValidationError("File content signature does not match allowed types.")
```

- **Priority:** HIGH
-

Issue 14 — Hardcoded Dialing Code Prefix Overriding International Phone Formats

- **Category:** Business Logic & Reliability
- **Severity:** Medium
- **File:** `core/services/sms_services.py`
- **Function/Class:** `SmsService.send_sms`
- **Line:** 20
- **Problem:** The SMS service unconditionally prepends India's dialing code +91 to all recipient mobile numbers.
- **Evidence:**

```
message = client.messages.create(
    body=template,
    from_=settings.TWILIO_PHONE_NUMBER,
    to=f"+91{mobile}"
)
```

- **Why this is a problem:** PartsHub is deployed in the United Arab Emirates (UAE), where all phone numbers use the +971 prefix, with Dubai timezone and AED currency. Hardcoding +91 prevents SMS messages from reaching UAE users and sends OTPs to unintended mobile subscribers in India.
- **Production impact:** All SMS-based verification and notifications in the primary market (UAE) fail completely or deliver confidential security codes to incorrect third parties.
- **Recommended fix:** Accept an E.164 formatted international phone number or dynamically prepend the user's registered `country_code` field:

```
destination = mobile if mobile.startswith("+") else f"{country_code}{mobile}"
message = client.messages.create(body=template,
    from_=settings.TWILIO_PHONE_NUMBER, to=destination)
```

- **Priority:** HIGH
-

Issue 15 — Excessive Database Roundtrips and Memory Buffering in `AuditTrailMiddleware`

- **Category:** Performance & Architecture
- **Severity:** High
- **File:** `core/middleware.py`
- **Function/Class:** `AuditTrailMiddleware.process_view, process_response, _resolve_user`
- **Line:** 115, 220–241, 274–284, 308–319
- **Problem:** On every mutating request (POST, PUT, PATCH, DELETE), the middleware executes redundant JWT authentication database lookups, consumes request bodies into RAM, runs snapshot queries, and performs synchronous DB logging.
- **Evidence:**

```
# Line 229: Re-runs JWT auth and DB queries in middleware
for authenticator_path in
("accounts.authentication.OrganizationJWTAuthentication", ...):
    authenticator = getattr(module, class_name)()
    result = authenticator.authenticate(request)

# Line 115: Buffers body
body = request.body.decode("utf-8").strip()

# Line 308: Synchronous DB write in request loop
log_activity(...)
```

- **Why this is a problem:**
 1. DRF will authenticate the user anyway during view processing; re-authenticating in Django middleware causes double database queries per request.
 2. Reading `request.body` consumes multipart form data and file streams into RAM.
 3. `_resolve_old_data` executes unindexed reflection queries before the view runs.
 4. Writing activity logs synchronously adds latency to the client response.
- **Production impact:** Adds 50–200ms of latency to every mutating HTTP request, doubles database authentication queries, and degrades server throughput under load.
- **Recommended fix:** Extract audit logging from generic middleware and place it into domain service events or signals. Offload activity log persistence to an asynchronous queue or execute it via `transaction.on_commit()`.
- **Priority:** HIGH

Issue 16 — Broken Soft Delete Abstraction and Misleading

Inheritance Pattern

- **Category:** Database & Architecture
- **Severity:** Medium
- **File:** accounts/models.py
- **Function/Class:** TimeStampedSoftDeleteModel
- **Line:** 29–35
- **Problem:** The abstract base class TimeStampedSoftDeleteModel contains only timestamp fields and defines no soft delete fields or methods.
- **Evidence:**

```
class TimeStampedSoftDeleteModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
```

- **Why this is a problem:** Inheriting models (such as OrganizationUser, BankDetails, UserAddressDetails) do not actually support soft delete. Developers assuming soft delete functionality is present call `.delete()`, causing hard cascading deletions of production data. Meanwhile, other models manually declare `is_deleted = models.BooleanField(default=False)` independently, leading to fractured deletion semantics across the database.
- **Production impact:** Accidental irreversible data loss and inconsistent query filtering across models.
- **Recommended fix:** Either rename the model to TimeStampedModel or properly implement soft delete fields and a custom model manager:

```
class SoftDeleteManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(is_deleted=False)

class TimeStampedSoftDeleteModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    is_deleted = models.BooleanField(default=False, db_index=True)
    deleted_at = models.DateTimeField(null=True, blank=True)

    objects = SoftDeleteManager()
    all_objects = models.Manager()

    def delete(self, using=None, keep_parents=False):
        self.is_deleted = True
        self.deleted_at = timezone.now()
        self.save(update_fields=["is_deleted", "deleted_at"])

    class Meta:
        abstract = True
```

- **Priority:** MEDIUM
-

Issue 17 — Dual User Identity Models and Incompatible Authentication Schemes

- **Category:** Architecture & Authentication
 - **Severity:** High
 - **File:** `accounts/models.py`, `admin_login/admin_authentication.py`, `core/permissions/admin_permissions.py`
 - **Function/Class:** `OrganizationUser` vs `django.contrib.auth.models.User`
 - **Line:** 470 (`accounts/models.py`), 1–16 (`admin_authentication.py`)
 - **Problem:** The application uses two separate, incompatible user models across the codebase: `accounts.OrganizationUser` (UUID primary key) and `django.contrib.auth.models.User` (integer primary key).
 - **Evidence:** `settings.REST_FRAMEWORK["DEFAULT_AUTHENTICATION_CLASSES"]` is set exclusively to `OrganizationJWTAuthentication`. Admin endpoints use `AdminJWTAuthentication` or `Django TokenAuthentication`. Permissions such as `HasGroupPermission` check `request.user.groups`, which raises `AttributeError` if evaluated on `OrganizationUser`.
 - **Why this is a problem:** Fragmenting the user concept into two disjoint models causes cross-module incompatibility. Admin tokens sent to core endpoints fail with `AuthenticationFailed("User not found")`. Conversely, organization user tokens sent to admin-gated endpoints crash with missing attribute errors.
 - **Production impact:** Frequent 500 errors, broken admin-to-tenant delegation workflows, and duplicated user administration overhead.
 - **Recommended fix:** Unify the user hierarchy using a custom Django user model (`AUTH_USER_MODEL`) that supports both platform administrative roles and multi-tenant organization memberships through explicit relations.
 - **Priority:** HIGH
-

Issue 18 — Unsafe Startup Auto-Migration Race Condition in Multi-Container Deployments

- **Category:** Deployment & Reliability
- **Severity:** High
- **File:** `docker-compose.yml`

- **Function/Class:** Container start command
- **Line:** 20–23
- **Problem:** The container entrypoint executes `python manage.py migrate` directly before starting the Gunicorn server in the application container.
- **Evidence:**

```
command: >
  sh -c "mkdir -p /app/uploads/documents /app/media &&
        chmod -R 775 /app/uploads /app/media &&
        python manage.py migrate &&
        gunicorn broker_platform.wsgi:application --bind 0.0.0.0:4288 --workers
3 --timeout 120 --access-logfile - --error-logfile -"
```

- **Why this is a problem:** In production environments running on AWS ECS, Kubernetes, or auto-scaled Docker hosts, multiple container instances launch simultaneously during a deployment. Having every web container run migrations concurrently triggers race conditions, migration lock contention, and schema corruption in PostgreSQL.
- **Production impact:** Failed deployments, deadlocks during rollouts, and potential schema corruption.
- **Recommended fix:** Decouple migrations from web server startup. Execute migrations as a dedicated single-shot pre-deployment release job (e.g. Kubernetes Job or ECS release task) before routing traffic to new container replicas.
- **Priority:** HIGH

Issue 19 — Container Execution as Root User in Production Environment

- **Category:** Security & Deployment
- **Severity:** High
- **File:** Dockerfile
- **Function/Class:** Container user configuration
- **Line:** 28–35
- **Problem:** The Docker container explicitly runs as the Linux `root` user in production.
- **Evidence:**

```
RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser
RUN mkdir -p /app/uploads/documents /app/media && chown -R appuser:appgroup /app

# Run as root so runtime chmod on mounted volumes works
# (needed for file uploads to /app/uploads and /app/media)
EXPOSE 8000
# Missing USER directive!
```

- **Why this is a problem:** Running production application containers as `root` violates the principle of least privilege. If an application vulnerability (such as remote code execution or file upload traversal) is exploited, the attacker obtains root access within the container, facilitating container escape and host compromise.
 - **Production impact:** Severe privilege escalation upon application compromise.
 - **Recommended fix:** Fix volume ownership during container initialization and add `USER appuser` before the runtime entrypoint. Store uploads in S3 (as configured) rather than local host volumes to avoid runtime permission mismatches.
 - **Priority:** HIGH
-

Issue 20 — Near-Zero Automated Test Coverage Across Critical Business and Security Domains

- **Category:** Testing & Reliability
- **Severity:** High
- **File:** `accounts/tests.py`, `claims/tests/`, `auctions/tests/`
- **Function/Class:** Entire test suite
- **Line:** 1–56 (`accounts/tests.py`)
- **Problem:** The test suite contains virtually no unit or integration tests for core authentication, authorization, multi-tenant isolation, or transactional financial logic.
- **Evidence:** In the entire `accounts` application (containing 3,763 lines of models, authentication controllers, KYC flows, and permission checks), the only test file (`accounts/tests.py`) contains a single 56-line test checking fallback strings in `_build_vehicle_details`.
- **Why this is a problem:** In a complex platform handling insurance claims, vehicle auctions, and payment transactions, the absence of automated tests means regression testing cannot be performed. Security flaws like Issue 01 (cross-tenant access) and Issue 02 (OTP response leak) exist undetected because no automated test validates tenant boundaries or response envelopes.
- **Production impact:** Frequent regression bugs, breaking API changes, and undetected security vulnerabilities introduced with each deployment.
- **Recommended fix:** Implement a comprehensive `pytest` test suite with high-priority test suites for:
 1. Tenant isolation (asserting Tenant A cannot read Tenant B claims).
 2. Authentication and OTP lifecycle (asserting OTP is not returned in responses).
 3. Bidding concurrency and auction status transitions.

- **Priority: HIGH**
-

Architecture Assessment

1. What architecture is currently being used?

The application is structured as a **Django monolithic REST API** with domain-divided applications (`accounts`, `claims`, `auctions`, `inventory`, `orders`, `job_cards`). However, the internal implementation resembles an **Active Record pattern with scattered procedural scripts** rather than a clean layered or domain-driven architecture.

2. Is it layered?

No, layering is superficial. While directories named `services`, `serializers`, and `views` exist, their responsibilities are blurred. Many views contain direct database queries, transactional boundaries, and serialization formatting. Meanwhile, service classes frequently accept raw DRF request objects and manipulate HTTP status codes directly.

3. Is business logic separated?

No. Business logic is fragmented across three distinct locations:

1. Inside DRF views (e.g. `InsuranceClaimOrderListView` filtering, sorting, and summarizing in-memory).
2. Inside serializers (e.g. `AdminRegistrationSerializer` creating groups and tokens in `.create()`, and `checklist_serializers.py` querying DB inside serializer fields).
3. Inside service classes.

4. Is database access separated?

No. There is no Repository layer or Data Access Object (DAO) pattern. Views, middleware, serializers, and services all make direct calls to `Model.objects.filter()`. Database-specific SQL (e.g. `SELECT nextval(...)`) is scattered across services and migrations.

5. Is authentication centralized?

No. Authentication is split between `OrganizationJWTAuthentication` (for org users) and `AdminJWTAuthentication/TokenAuthentication` (for admins). In addition, `AuditTrailMiddleware` implements its own ad-hoc JWT authentication logic by manually re-parsing tokens on every mutating request.

6. Is authorization centralized?

No. Authorization is severely fragmented. Some views use DRF permission classes (`IsInsuranceAdmin`), some check `request.auth.get("hq_or_branch")` inline, only one module (`job_cards`) uses `check_object_permissions`, and several views omit

permission classes entirely.

7. Is validation centralized?

No. Input validation is inconsistently split between DRF serializers, custom service validator methods (e.g., `_validate_deal_state`), and utility functions. Validation error responses differ in structure across modules.

8. Is error handling centralized?

No. `core/exception_handler.py` only intercepts `SuspiciousOperation` and falls back to DRF's default handler. Individual views wrap logic in bespoke `try/except` blocks returning custom `error_response` dictionaries, while unhandled exceptions escape to Django's WSGI handler as HTML 500 error pages.

9. Is the application modular?

Low modularity. Although separated into Django apps, apps are tightly coupled via circular model dependencies, foreign keys across domain boundaries, and ad-hoc imports inside function bodies to prevent import errors at startup.

10. Is it suitable for enterprise scale?

No. The combination of cross-tenant IDOR vulnerabilities, blocking synchronous integrations, in-process multi-worker scheduling, in-memory query processing, and absence of automated test suites makes the current implementation unsuitable for enterprise production.

Python API Quality Assessment

When evaluated against modern enterprise Python/Django standards:

1. Django REST Framework Usage:

- **Underutilization of Generic Views:** Standard DRF patterns (`ModelViewSet`, `GenericViewSet`, `get_queryset`, `get_object`) are bypassed in favor of raw `APIView` classes implementing manual `.filter().first()` queries.
- **Serializer Anti-Patterns:** Serializers perform database queries during field evaluation (`SerializerMethodField`), inducing N+1 query performance degradation.
- **Response Consistency:** Failure to standardize on DRF response envelopes or utilize renderers.

2. Async / Concurrency:

- Entirely synchronous WSGI codebase running on Gunicorn with only 3 worker processes.
- Zero `async/await` adoption for high-latency external I/O (Twilio, Pusher, S3

uploads, SMTP).

3. Database & ORM:

- Absence of database connection pooling (psycopg2-binary without PgBouncer or persistent pool configuration).
- Inconsistent soft-delete modeling.
- Locking records during external HTTP calls (SELECT FOR UPDATE + Pusher triggers).

NestJS Comparison

Area	Current Python API	Proper NestJS Architecture	Problem in Current Implementation
Modules	Django apps with loose boundaries and frequent inline circular imports	Strict NestJS <code>@Module()</code> encapsulating providers and exporting only necessary services	High coupling; hard to isolate or test features independently
Controllers	Monolithic <code>APIView</code> classes containing business logic, DB queries, and formatting	Slim <code>@Controller()</code> responsible purely for HTTP routing, status codes, and DTO parsing	Business logic tightly coupled to HTTP request lifecycle
Services	Procedural static method classes mixing DB operations with HTTP response creation	Injected <code>@Injectable()</code> domain services handling pure business logic	Difficult to mock dependencies and unit test business logic
Repository	No repository layer; direct ORM calls (<code>Model.objects</code>) scattered across all layers	TypeORM / Prisma Repositories or custom Repository pattern abstracting queries	Queries cannot be centrally optimized or substituted
DTO Validation	DRF Serializers with inconsistent validation methods and partial field checks	<code>class-validator</code> + <code>ValidationPipe</code> with strict type validation before controller execution	Invalid data enters service layers; MIME types are bypassed
Authentication	Dual fragmented JWT authenticators (<code>OrganizationJWT</code> vs <code>AdminJWT</code>)	Centralized <code>@UseGuards(JwtAuthGuard)</code> using <code>@nestjs/passport</code>	Admin and Org tokens incompatible; duplicate auth checks in middleware
Authorization	Fragmented permission	Declarative	Severe cross-tenant

authentication/	
guards/	# Universal JWT Auth Guard
strategies/	# Token validation & token-version
checks	
decorators/	# @CurrentUser(), @CurrentTenant()
authorization/	
guards/	# RBAC & Tenant Isolation Guards
policies/	# Object-level access policies (CASL /
Rule engine)	
database/	
base.model.ts / base.py	# Standardized timestamps & soft-
delete managers	
repository.base.ts	# Abstract repository interface
filters/	# Global Exception Filter (enforcing
standard envelope)	
interceptors/	# Logging & Request Correlation ID (X-
Correlation-ID)	
pipes/	# Strict DTO Validation (class-
validator / Pydantic)	
utils/	
file-security/	# Magic-byte MIME & antivirus
inspection	
modules/	# Self-contained business domain
modules	
identity/	# Unified Users, Roles, Permissions,
Auth & OTP	
controllers/	
services/	
repositories/	
dto/	
tenancy/	# Organizations, Branches, KYC &
Verification	
controllers/	
services/	
repositories/	
claims/	# Claim Lifecycle & Workflow State
Machine	
controllers/	
services/	
repositories/	
dto/	
auctions/	# Real-Time Bidding & Auction
Lifecycle	
controllers/	
services/	# Atomic bidding with decoupled event
dispatch	
repositories/	
orders-invoices/	# Garage & Supplier Orders, Invoicing
inventory/	# Parts, Lots, Fitments, Hot Deals
job-cards/	# Garage Repair Tracking & Evidence
integrations/	# Decoupled Third-Party Adapters
sms/	# Twilio adapter with E.164 formatting
& circuit breaker	
email/	# Mail service with template engine &
async dispatch	
realtime/	# Pusher adapter (event-driven)
storage/	# S3 signed proxy & presigned URL
generator	

```
|   └─ payments/                                # Real Payment Gateway Adapters
|   (Stripe / Telr)
|   └─ workers/                                # Decoupled Background Task Workers
|       └─ queue/                             # BullMQ / Celery worker definitions
|           └─ jobs/                         # Bulk import, license expiry, status
transitions
```

Final Verdict

1. Current Architecture Level:

Small application / Prototype

(Contains extensive domain features, but lacks the isolation, tenancy security, resilience, and testing required for enterprise production).

2. Biggest Architectural Problem:

Lack of Layered Separation and Tenancy Enforcement: Business logic, security controls, and database queries are tightly tangled in views, resulting in systemic bypasses of tenant isolation.

3. Biggest Security Problem:

Direct Password Reset Account Takeover (Issue 02) & Cross-Tenant Data Access (Issue 01):

The plain OTP is returned directly in HTTP response bodies, and any HQ user can inspect and modify claims belonging to any organization.

4. Biggest Scalability Problem:

In-Memory Query Slicing and In-Process Background Workers: Loading entire tables into Python memory for serialization and running background scheduler threads inside web workers will cause server deadlocks and memory crashes under high concurrent load.

5. Biggest Maintainability Problem:

Near-Zero Test Coverage & God Models: Having 3,763 lines in a single model file (`accounts/models.py`) with virtually no unit or integration tests makes refactoring and extending features extremely risky.

6. Biggest Database Problem:

Pessimistic Row Locks Held During Synchronous External HTTP Requests: Bidding operations hold `SELECT FOR UPDATE` database locks while waiting for third-party Pusher HTTP responses.

7. Biggest Reliability Problem:

Synchronous External Dependencies: Inline synchronous calls to Twilio, Pusher, and SMTP without timeouts will exhaust the 3 Gunicorn workers during external vendor hiccups.

8. What MUST be fixed BEFORE production?

- **Issue 01:** Patch all claims endpoints to strictly scope queries by `organization_id__in=expand_org_scope(request.user)`.
- **Issue 02:** Remove OTP from `ForgotPasswordView` response.
- **Issue 03:** Hash OTPs in the database and remove `print()` statements.
- **Issue 04:** Fix OTP session replay in admin password reset and invalidate existing tokens.
- **Issue 05:** Reduce admin access token lifetime from 30 days to 30 minutes.
- **Issue 06 & 07:** Enforce `IsAuthenticated` by default and remove wildcard defaults for CORS, Allowed Hosts, and webhooks.
- **Issue 09:** Move Pusher notifications outside of `@transaction.atomic` using `transaction.on_commit()`.
- **Issue 10:** Disable in-process `APScheduler` in Gunicorn web workers.
- **Issue 13 & 14:** Fix magic byte file validation and remove hardcoded +91 dialing prefix.
- **Issue 19:** Switch Docker container execution to unprivileged user (`appuser`).

9. What can be fixed later?

- Transitioning from synchronous email/SMS to a full asynchronous Celery/BullMQ worker cluster.
- Refactoring `TimeStampedSoftDeleteModel` and splitting `accounts/models.py` into modular domain entities.
- Full unification of `OrganizationUser` and `Django auth.User`.
- Replacing PostgreSQL sequence DDL with platform-agnostic sequence generators.

10. Strategic Recommendation:

Architecturally Redesign and Incrementally Rewrite.

The core domain models and business state machines (e.g. `HotDealStatusMachine`, `AuctionBidService`, verification checklists) reflect genuine business requirements and can be preserved. However, the controller layer, authentication/authorization layer, background job pipeline, and database query layers require an immediate architectural overhaul to eliminate critical security holes and scalability bottlenecks before enterprise launch.