High-Level Design - Multi-Tenant Django SaaS

Scope: Multi-tenancy, Admin, Public/Private schema, Redis cache, Elasticsearch, WebSockets

Architecture Objective

Build a scalable SaaS platform where each tenant is isolated at database-schema level, while still sharing core platform controls in a public schema. The design enforces tenant boundaries across HTTP APIs, cache keys, websocket channels, and Elasticsearch indexes.

Technology Stack

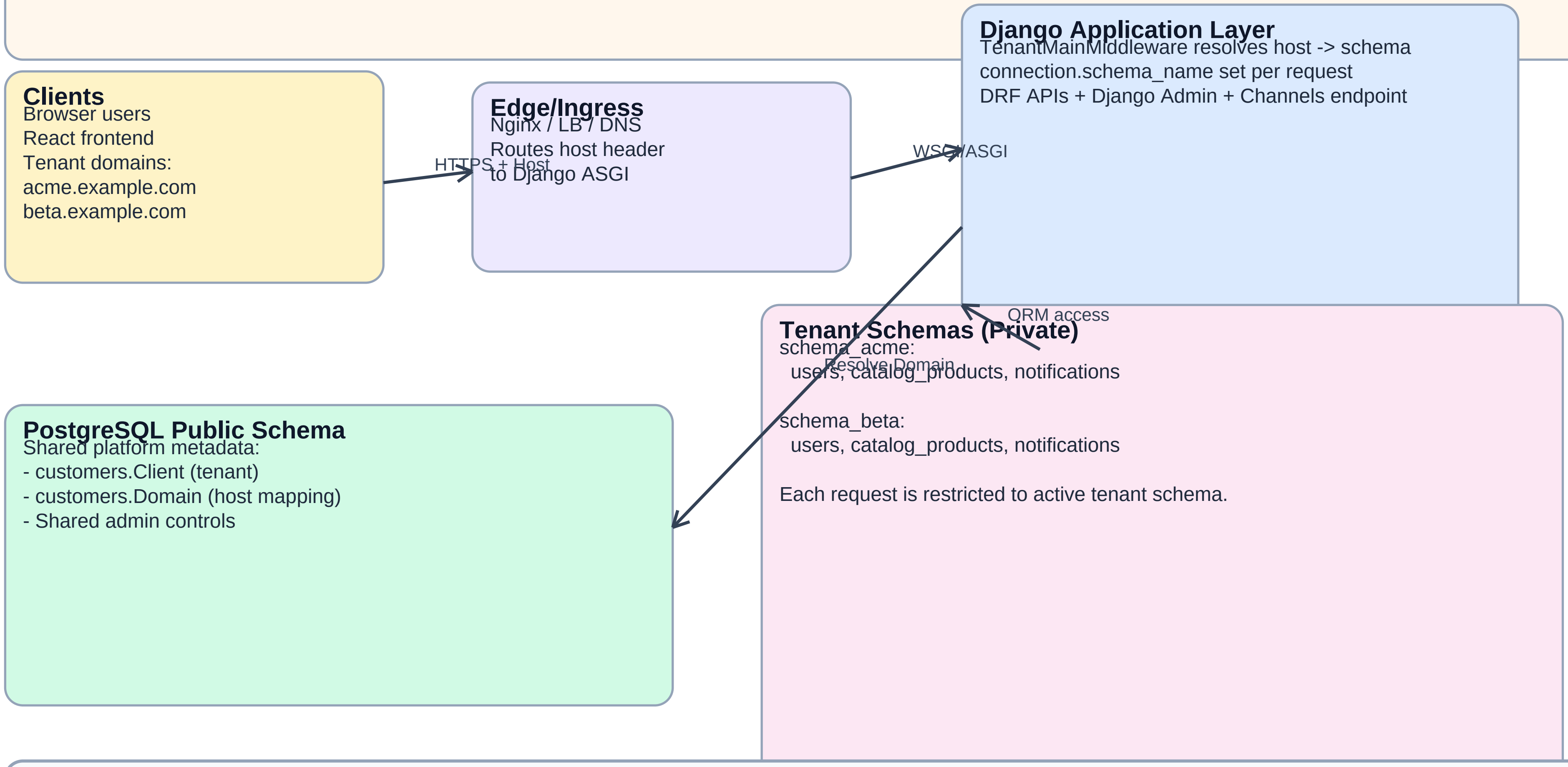
Django + django-tenants
Django REST Framework for CRUD APIs
PostgreSQL schemas for tenant isolation
Redis for cache + channel layer
Elasticsearch official Python client for search
Django Channels + React websocket client

Requirement	Design Choice	Primary Components
1. Multi-Tenant	Domain-driven schema resolution via django-tenants middleware.	Client/Domain models, TenantMainMiddleware, PostgreSQL schema search path.
2. Admin Panel	Public admin for tenant lifecycle, tenant admin for tenant-local entities.	ClientAdmin, DomainAdmin, UserAdmin, ProductAdmin, NotificationAdmin.
3. Public/Private + Redis	Shared data in public schema, tenant data in private schema, cache-aside strategy.	SHARED_APPS, TENANT_APPS, Redis cache keys scoped by schema.
4. Elasticsearch	Per-tenant index naming and signal-based sync for write consistency.	ProductSearchService, post_save/post_delete signals, index pattern by schema.
5. WebSockets	Authenticated channels with schema-scoped user groups.	Channels consumer, Redis channel layer, React live notification feed.

1) Multi-Tenant Architecture

Tenant resolution, schema isolation, and request routing

Requirement 1 (From Assignment)
Implement a multi-tenant architecture where each tenant has its own isolated database schema. Use Django routing capabilities or a library like django-tenant-schemas.



Clients
Browser users
React frontend
Tenant domains:
acme.example.com
beta.example.com

Edge/Ingress
Nginx / LB / DNS
Routes host header
to Django ASGI

Django Application Layer
TenantMainMiddleware resolves host -> schema
connection.schema_name set per request
DRF APIs + Django Admin + Channels endpoint

PostgreSQL Public Schema
Shared platform metadata:
- customers.Client (tenant)
- customers.Domain (host mapping)
- Shared admin controls

Tenant Schemas (Private)
schema_acme:
  users, catalog_products, notifications

schema_beta:
  users, catalog_products, notifications

Each request is restricted to active tenant schema.

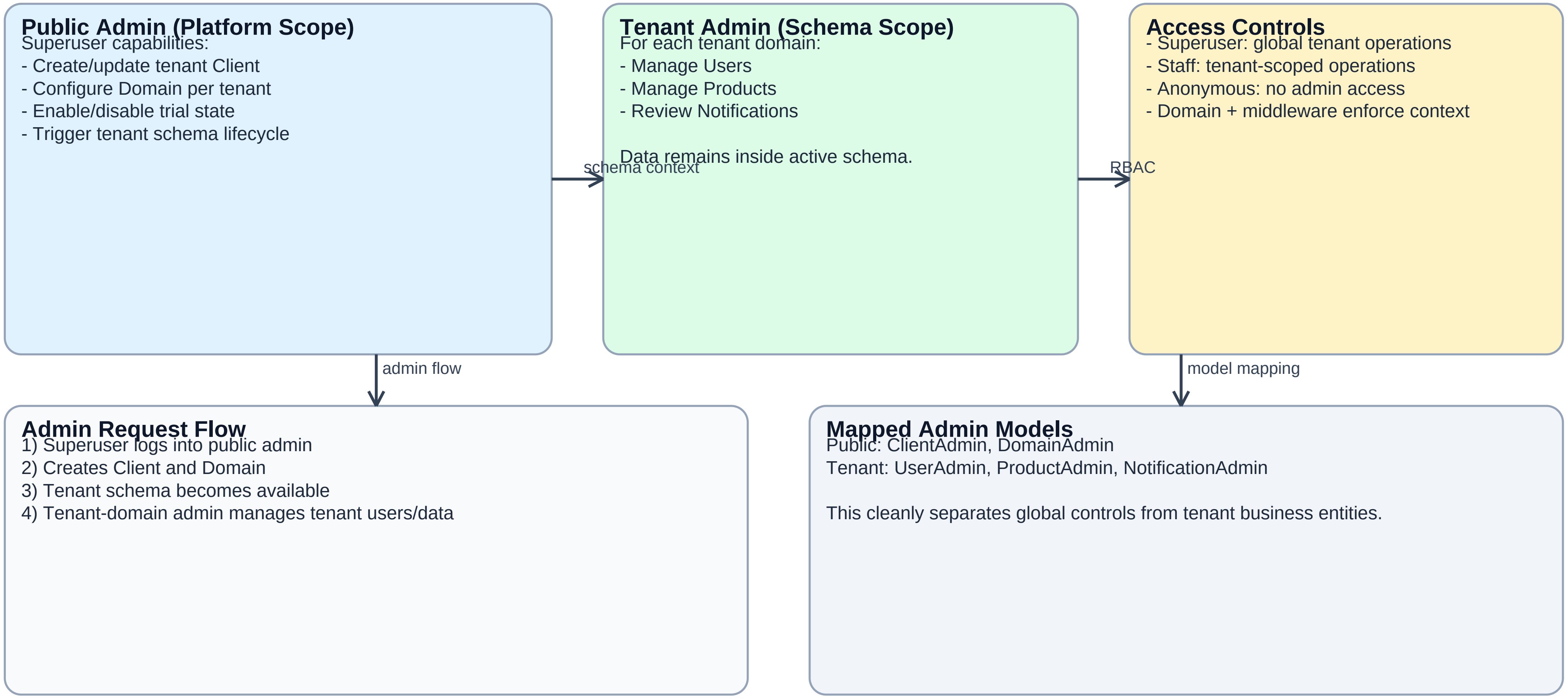
Key HLD Decisions
Use schema-per-tenant for hard data isolation. Public schema only stores tenant registry and shared controls. Tenant context must be propagated to HTTP, cache, websocket groups, and search index naming.

2) Django Admin Panel

Multi-tenant administration model aligned to superuser governance

Requirement 2 (From Assignment)

Customize the Django admin panel to support multi-tenancy. Ensure that superusers can manage tenants and their users through the admin panel.



Public Admin (Platform Scope)

Superuser capabilities:
- Create/update tenant Client
- Configure Domain per tenant
- Enable/disable trial state
- Trigger tenant schema lifecycle

Tenant Admin (Schema Scope)

For each tenant domain:
- Manage Users
- Manage Products
- Review Notifications

Data remains inside active schema.

Access Controls

- Superuser: global tenant operations
- Staff: tenant-scoped operations
- Anonymous: no admin access
- Domain + middleware enforce context

Admin Request Flow

1) Superuser logs into public admin
2) Creates Client and Domain
3) Tenant schema becomes available
4) Tenant-domain admin manages tenant users/data

Mapped Admin Models

Public: ClientAdmin, DomainAdmin
Tenant: UserAdmin, ProductAdmin, NotificationAdmin

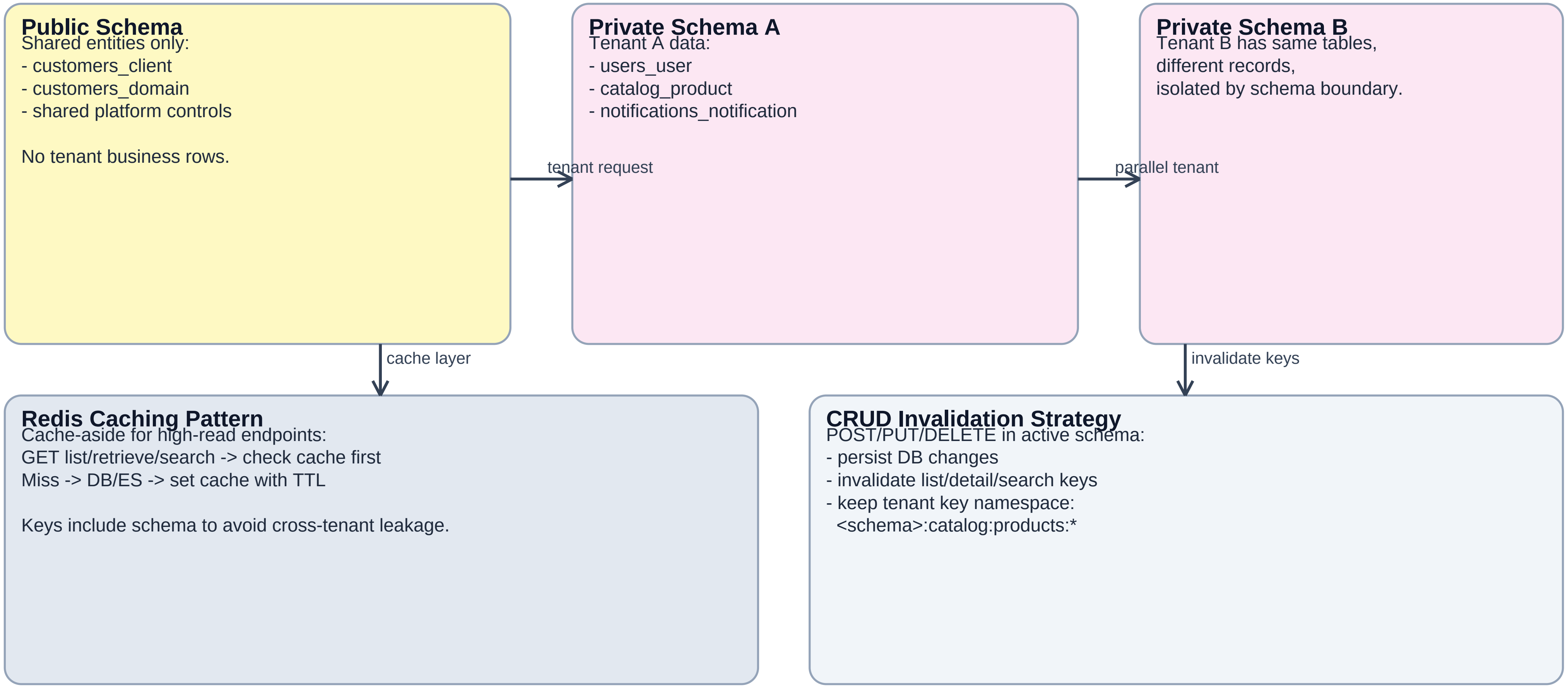
This cleanly separates global controls from tenant business entities.

3) Public & Private Schema with Redis Cache

Schema boundary design and cache strategy for CRUD operations

Requirement 3 (From Assignment)

Create a public schema for shared data and a private schema for tenant-specific data. For CRUD operations, implement Redis cache wherever possible.



Public Schema

Shared entities only:
- customers_client
- customers_domain
- shared platform controls

No tenant business rows.

Private Schema A

Tenant A data:
- users_user
- catalog_product
- notifications_notification

Private Schema B

Tenant B has same tables,
different records,
isolated by schema boundary.

Redis Caching Pattern

Cache-aside for high-read endpoints:
GET list/retrieve/search -> check cache first
Miss -> DB/ES -> set cache with TTL

Keys include schema to avoid cross-tenant leakage.

CRUD Invalidation Strategy

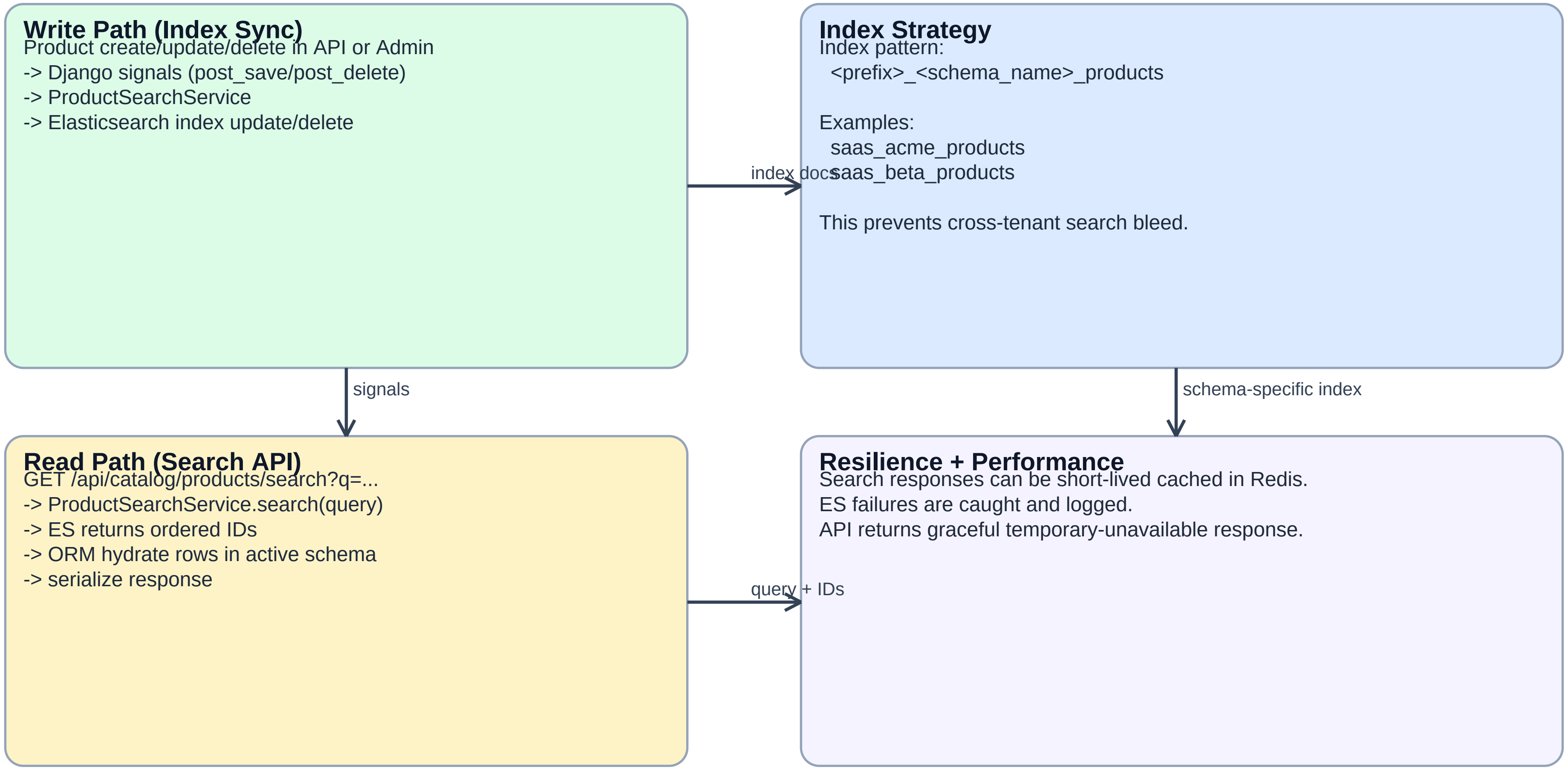
POST/PUT/DELETE in active schema:
- persist DB changes
- invalidate list/detail/search keys
- keep tenant key namespace:
  <schema>:catalog:products:*

4) Elasticsearch Architecture

Tenant-isolated indexing and search query path

Requirement 4 (From Assignment)

Integrate Elasticsearch using the official Elasticsearch Python client. Index and search data across tenants while keeping each tenant's data isolated.



Write Path (Index Sync)

Product create/update/delete in API or Admin
-> Django signals (post_save/post_delete)
-> ProductSearchService
-> Elasticsearch index update/delete

Index Strategy

Index pattern:
  <prefix>_<schema_name>_products

Examples:
  saas_acme_products
  saas_beta_products

This prevents cross-tenant search bleed.

Read Path (Search API)

GET /api/catalog/products/search?q=...
-> ProductSearchService.search(query)
-> ES returns ordered IDs
-> ORM hydrate rows in active schema
-> serialize response

Resilience + Performance

Search responses can be short-lived cached in Redis.
ES failures are caught and logged.
API returns graceful temporary-unavailable response.

Security Note

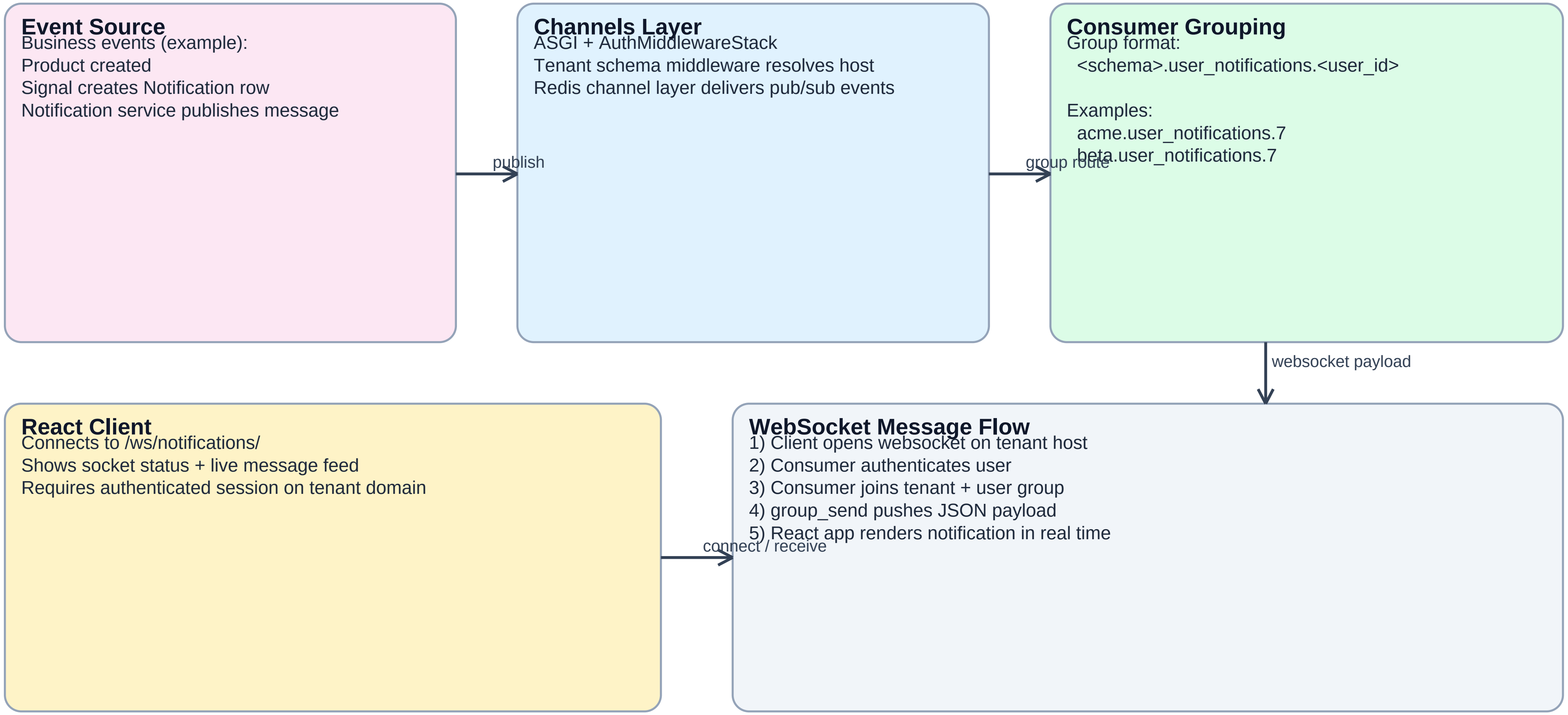
Never query a shared cross-tenant index for tenant endpoints. Always derive index name from current schema context.

5) WebSockets Notifications Architecture

Real-time push with Channels and tenant-safe group names

Requirement 5 (From Assignment)

Implement WebSockets using Django Channels for real-time notifications and a simple React app that connects to the WebSocket server and displays notifications.



Event Source

Business events (example):
Product created
Signal creates Notification row
Notification service publishes message

Channels Layer

ASGI + AuthMiddlewareStack
Tenant schema middleware resolves host
Redis channel layer delivers pub/sub events

Consumer Grouping

Group format:
  <schema>.user_notifications.<user_id>

Examples:
  acme.user_notifications.7
  beta.user_notifications.7

React Client

Connects to /ws/notifications/
Shows socket status + live message feed
Requires authenticated session on tenant domain

WebSocket Message Flow

1) Client opens websocket on tenant host
2) Consumer authenticates user
3) Consumer joins tenant + user group
4) group_send pushes JSON payload
5) React app renders notification in real time

Isolation Guarantee

Even if two tenants have user_id=7, schema-prefixed group names prevent cross-tenant notification leakage.

Additional Considerations

Security, error handling, tests, documentation, and deployment

Security

- Authenticated DRF endpoints
- Authenticated websocket connections
- Secure cookie toggles for prod
- Tenant-scoped cache keys, groups, indexes
- Admin access control by role

Reliability

- Structured logging
- Graceful ES failure handling
- Cache fallback to DB on misses
- Redis channel layer decouples producers/consumers

Testing Strategy

- Unit tests for models/services
- API auth and schema behavior tests
- Signal/index sync tests
- Websocket group naming tests

Deployment Topology

Containers/Services:

1) Django ASGI service
2) PostgreSQL (schema-per-tenant)
3) Redis (cache + channels)
4) Elasticsearch
5) Frontend (React/Vite static client)

CI/CD should run migrations, tests, linting, and security checks.

Documentation Deliverables

Submission should include:

- Setup instructions
- Tenant bootstrap steps (Client + Domain)
- API and websocket endpoints
- HLD diagrams and design decisions
- Test execution guide
- Known constraints and production hardening checklist

Evaluation Alignment

The above design directly maps to adherence, code quality, scalability/performance, and documentation clarity criteria.