International Science Journal of Engineering & Agriculture
2026; 5(3): 107-123
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260503.10
ISSN: 2720-6319

INTERNATIONAL
SCIENCE GROUP

Architectural patterns for full-stack TypeScript web applications with Next.js and NestJS

Igor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Bohdan Kupyra
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0007-4395-3337

Abstract: Modern web application development has converged on TypeScript as the foundational language of both the user-facing presentation layer and the data-serving backend, enabling unified development practices, shared type definitions across the network boundary, and substantially reduced cognitive overhead during cross-stack engineering work. However, the architectural patterns for combining these layers vary widely across the industry, and the selection of frameworks, orchestration approaches, and data access strategies significantly affects developer productivity, application performance, and long-term maintainability. This research examines the architectural design and implementation of a full-stack TypeScript web application using Next.js version 15 with the App Router on the frontend and NestJS version 11 with the Prisma object-relational mapper on the backend, deployed against a SQLite relational database. The application implements a customer-facing e-commerce platform with 29 user interface routes, 59 REST API endpoints, and 20 database models, providing a representative scale for evaluating architectural choices. The study analyzes the comparative benefits of hybrid server-side and client-side rendering enabled by React Server Components, the role of the NestJS dependency injection container in producing modular and testable backend code, the type-safe data access guarantees offered by Prisma through schema-first code generation, and the operational implications of organizing both packages within a single npm workspace monorepo. Empirical measurements demonstrate that the proposed architecture achieves a 164 kilobyte First Load JavaScript footprint on the home page through aggressive code splitting and Server Components, sub-200 millisecond response times for catalog queries through composite database indexing, and a deployment artifact of approximately 200 megabytes including application code and Node.js runtime. The research identifies architectural patterns that contribute to this performance profile, including the segmentation of routes into authenticated and public groups through route groups, the centralization of authentication state in React Context providers, the use of repository-like services with globally injected Prisma access, and the colocation of validation rules between client and server through structurally aligned schemas. The proposed architecture serves as a reference implementation for small-to-medium TypeScript-first web applications and as an educational template for software engineering curricula.

Keywords: full-stack TypeScript; Next.js; NestJS; Prisma ORM; App Router; React Server Components; dependency injection; web application architecture; monorepo; e-commerce

1. Introduction

The architecture of web applications has undergone substantial revision over the past decade, driven by the parallel evolution of browser capabilities, mobile device penetration, and developer expectations for productivity and type safety. The dominant pattern of the 2000s—a server-rendered monolith built around a relational database, a templating engine, and progressive enhancement

through JavaScript—has given way to a family of patterns in which the user interface is increasingly handled by sophisticated client-side frameworks while the server provides data through documented application programming interfaces. This separation enables independent scaling of the presentation and data layers, supports the development of native mobile applications against the same backend, and simplifies the introduction of additional consumer applications such as administrative dashboards or third-party integrations.

The technological choices available for constructing such applications have multiplied substantially, and the selection of an appropriate stack now constitutes a substantive architectural decision in its own right. On the frontend, the major options include React-based meta-frameworks (Next.js, Remix), Vue-based meta-frameworks (Nuxt), the Svelte-based SvelteKit, and the Angular framework with its associated tooling. Each combines a component model, a routing system, and varying degrees of integration with server-side rendering and incremental hydration. On the backend, the choices include Node.js frameworks (Express, Fastify, NestJS), Python frameworks (Django, FastAPI), Ruby frameworks (Rails, Sinatra), Go frameworks, and Rust frameworks (Actix, Axum). The selection of each layer is partially independent, but combinations that share a language, particularly TypeScript across both layers, offer practical advantages in code reuse, type sharing, and reduced cognitive load during cross-stack work.

The combination of Next.js on the frontend and NestJS on the backend has emerged as a particularly compelling pairing for TypeScript-first applications. Next.js [1] is the leading React-based meta-framework, offering server-side rendering, static generation, and client-side hydration within a unified routing model, with native support for the React Server Components paradigm introduced in React 19 [2]. NestJS [3] is a structured backend framework that imposes a dependency-injection-based modular architecture inspired by the Angular framework on the frontend, offering decorator-driven controllers, services, and guards, together with first-class support for OpenAPI documentation, validation, and testing. The two frameworks share a strict TypeScript convention, similar levels of community maturity, and a comparable approach to file-system-driven organization, making their combination particularly natural.

The Prisma object-relational mapper [4] complements this stack by providing schema-first type-safe data access. A single Prisma schema file defines the database structure declaratively; the Prisma command-line tool generates TypeScript types from this schema and synchronizes the database through automatically created migrations. Code that uses the generated client benefits from full IDE autocompletion of available models, fields, and relations, and from compile-time errors on any access that does not match the schema. The combination of TypeScript strict mode and Prisma's generated types produces a development experience in which entire categories of runtime errors—null reference exceptions, type confusion between numbers and strings, accidental access of unloaded relations—are surfaced before code execution.

Although each of these technologies is individually well-documented, the literature contains comparatively few integrated treatments that examine how they combine into a coherent architecture, what design patterns emerge from their use together, and what performance characteristics result. Existing tutorials typically focus on the rapid assembly of a working application without examining the architectural decisions that govern its long-term maintainability, while research articles tend to address individual subsystems—authentication, rendering, data access—in isolation. The present research aims to fill this gap by examining a complete application implementation and identifying the architectural patterns that contribute to its observed performance, modularity, and maintainability characteristics.

The reference application analyzed in this research is a customer-facing e-commerce platform for the floral retail domain. The choice of domain is incidental to the architectural analysis but provides a realistic context with a representative range of functional requirements: a product catalog with filtering and search, user accounts with authentication and authorization, shopping cart management with persistence across sessions, order placement with status transitions, administrative content management, and email notifications. The application comprises 53 TypeScript source files

on the frontend and 58 on the backend, totaling approximately 6,300 lines of application code excluding tests, configuration, and generated artifacts, and is deployed through a Docker Compose specification that brings the entire system up with a single command.

2. Object and subject of research

The object of this research is the architecture of modern full-stack web applications that combine TypeScript on both the presentation and data layers and that follow a decoupled client-server pattern in which the user interface and the application programming interface are independently deployable services. Such architectures are characteristic of contemporary commercial web platforms—e-commerce sites, software-as-a-service applications, internal enterprise tools, and content management systems—where the decoupling enables independent scaling, multi-channel consumption of the same backend, and the introduction of mobile or partner-facing clients without modification of the core data layer.

The object encompasses both the static structure of such applications—the organization of modules, components, services, and data models—and their dynamic behavior under typical user interactions, including page navigation, form submission, asynchronous data fetching, error handling, and authentication state changes. It also includes the operational characteristics that emerge from the chosen architecture: deployment artifact size, application startup time, runtime memory consumption, request latency under various access patterns, and the developer experience of working with the codebase as it scales in size and team membership.

The subject of this research is the set of architectural patterns and design choices that govern the construction of full-stack TypeScript applications using Next.js for the frontend, NestJS for the backend, and Prisma for data access. The investigation focuses on patterns that are sufficiently general to apply to a wide range of application domains while remaining concrete enough to admit empirical evaluation in a specific implementation. Specifically, the research examines patterns for the organization of frontend routes, the integration of authentication state, the layering of validation between client and server, the use of dependency injection in backend modules, the role of database indexing in achieving query performance, and the workflow of generating and applying database migrations through schema-driven tooling.

The subject explicitly excludes considerations that are highly application-specific—the business logic of any particular domain, the visual design of user interfaces, the choice of authentication identity providers—and focuses on the structural and operational properties that generalize across application domains. Although the reference application is in the e-commerce domain, the patterns examined are equally applicable to administrative dashboards, customer-facing service portals, internal data management tools, and other classes of business application that share the underlying technical structure.

3. Target of research

The principal target of this research is the formulation and empirical validation of a reference architecture for full-stack TypeScript web applications that demonstrably achieves three objectives: a development experience that minimizes the cognitive overhead of cross-stack work through unified language and shared type discipline, an operational profile suitable for deployment on commodity infrastructure without specialized scaling techniques, and a code organization that supports the introduction of new features and team members without proportional growth in complexity.

Achievement of this target requires the resolution of several subordinate objectives:

– Specification of the integration pattern between Next.js and NestJS, including the handling of cross-origin resource sharing, the transmission of authentication credentials between the two services, and the synchronization of TypeScript types when they are not literally shared between packages.

– Definition of route organization patterns within the Next.js App Router that segment the user interface into logical groups—public pages, authenticated user pages, administrative pages—with distinct layout, authentication, and authorization requirements.

– Analysis of the role of React Server Components in producing minimal client-side JavaScript bundles, including the identification of components that benefit from server rendering and those that require client-side interactivity.

– Specification of the NestJS module organization pattern, including the use of global services for cross-cutting concerns, the use of feature modules for domain-specific functionality, and the integration of guards and pipes into the request processing pipeline.

– Definition of the Prisma schema design pattern, including the relationship between database models and TypeScript types, the use of indices for query performance, and the migration workflow for schema evolution.

– Empirical validation of the proposed architecture through measurement of First Load JavaScript size, server response time, database query performance, deployment artifact size, and qualitative observations on the developer workflow.

4. Literature analysis

The literature surrounding full-stack TypeScript web application development spans framework documentation, design pattern catalogs, performance studies, and reflections on the evolution of the JavaScript ecosystem. The official documentation of Next.js [1] provides the most comprehensive treatment of the App Router paradigm introduced in version 13 and refined through version 15, including the patterns for nested layouts, route groups, parallel routes, and intercepting routes that together enable sophisticated user interface organization. The corresponding documentation of React 19 [2] introduces the Server Components paradigm and its implications for the boundary between server-rendered and client-rendered content.

The NestJS documentation [3] specifies the module system, dependency injection container, and decorator-based controllers that distinguish the framework from less opinionated Node.js alternatives such as Express. The framework's design draws heavily on the architectural patterns established by Angular [5], applied to the backend context rather than the frontend. Comparative studies of Node.js backend frameworks [6] generally place NestJS in the category of structured frameworks alongside Loopback and TypeORM-based stacks, distinguishing them from minimal middleware-driven frameworks such as Express and Fastify.

Prisma's documentation [4] specifies the schema language, the migration workflow, and the generated client API. The choice of Prisma over alternative TypeScript-friendly ORMs—including TypeORM [7], Drizzle ORM [8], and Sequelize—reflects trade-offs between schema-first declarative modeling (Prisma), TypeScript-first programmatic modeling (Drizzle), and decorator-based annotation of entity classes (TypeORM). The Prisma approach has been criticized for its requirement of a code generation step in the build process [9], but praised for the clarity of its schema language and the consistency of its generated types.

The Tailwind CSS framework, used in the reference application's styling layer, is documented extensively in its official sources [10]. The introduction of Tailwind version 4 in 2024 brought substantial architectural changes, including a CSS-first configuration model that eliminates the previous JavaScript-based configuration file and a switch to the Lightning CSS compiler [11] for build-time processing. The interaction of Tailwind with React Server Components requires no special handling because the framework operates entirely at compile time, generating a single CSS file consumed identically by server-rendered and client-rendered components.

TanStack Query [12], formerly known as React Query, has emerged as the dominant solution for managing server-derived state in React applications, offering automatic caching, request deduplication, background refetching, and optimistic updates. The library is widely regarded as superior to general-purpose state managers such as Redux for the specific case of server data, since

it incorporates domain knowledge about the semantics of cached data—freshness, invalidation, and refetching—that general state managers cannot express. Its integration with Next.js requires consideration of the boundary between server and client rendering, since queries can be prefetched on the server and hydrated on the client to avoid duplicate fetches.

Form management in modern React applications is dominated by React Hook Form [13] and Formik, with the former preferred for performance-sensitive applications due to its uncontrolled component approach. Pairing React Hook Form with the Zod schema validation library [14] provides type-safe form validation in which a single Zod schema describes both the runtime validation rules and the TypeScript type of the form data. This pattern, sometimes called "validation as types," has become a de facto standard in TypeScript-first applications and contributes substantially to the developer experience by eliminating duplicate type and validation declarations.

Containerization of full-stack applications is treated extensively in the Docker documentation [15] and in textbooks on cloud-native deployment. The multi-stage Dockerfile pattern, in which one stage compiles the application and a separate stage produces the runtime image with only the necessary artifacts, has become standard practice for Node.js applications. Combined with Docker Compose for multi-service orchestration, this approach produces self-contained deployment specifications that can be brought up on any host with a Docker installation, supporting the operational simplicity that has become a core selling point of TypeScript-based stacks.

Despite the breadth of documentation for individual technologies, the literature contains relatively few integrated treatments that examine how these components combine into a production-grade application architecture. Tutorial-oriented materials tend to demonstrate the assembly of a working prototype without examining the architectural decisions that govern long-term maintainability, while academic papers tend to focus on isolated technical contributions—a new caching strategy, a new rendering algorithm—without examining their integration with the surrounding stack. The present research addresses this gap by presenting an integrated architectural analysis of a complete deployed application.

5. Research methods

The methodology of this research combines architectural analysis, comparative evaluation against alternative technology choices, and empirical performance measurement. The analytical component identifies the architectural patterns embedded in the reference application and articulates the reasoning that motivated each pattern, drawing on the documentation of the underlying technologies, accepted industry practice, and the constraints imposed by the application's functional requirements. The comparative component contrasts each architectural choice with two or three commonly available alternatives, examining the trade-offs that informed the selection. The empirical component reports measurements collected from the deployed application under representative load conditions, with attention to the operational characteristics most relevant to the architectural decisions under examination.

The reference application was developed iteratively over a period of approximately two months by a single developer, with version control through Git and a commit history reflecting the major implementation milestones. The development followed a structured sequence: project scaffolding, backend skeleton with authentication, frontend skeleton with public pages, backend feature modules with API endpoints, frontend feature pages with API integration, administrative interface, automated tests, and operational packaging. The methodology benefits from this developmental sequence by enabling discussion of the architectural decisions in the order they were encountered and by providing concrete evidence of their long-term consequences.

Comparative evaluation was conducted through structured documentation review of the alternative technologies. For each architectural choice, two or three alternatives were identified from the documentation of the most widely deployed competing solutions, and the trade-offs were characterized in terms of dimensions relevant to the application context: type safety, runtime

performance, development velocity, deployment complexity, operational simplicity, and community maturity. The evaluation explicitly excluded performance benchmarking of alternative implementations, on the grounds that such benchmarks are highly sensitive to implementation details and would not yield generalizable conclusions about architectural choices.

Empirical measurement was conducted on a developer workstation with Node.js 22 LTS, with the application running in production mode after standard build optimizations. Frontend metrics were obtained from the Next.js build output, which reports the size of each generated JavaScript chunk and the per-route First Load JavaScript size. Backend metrics were obtained from instrumentation of representative endpoints with timing markers around the principal phases of operation, including database query execution, request validation, business logic, and response serialization. Database query performance was assessed through the use of the SQLite EXPLAIN QUERY PLAN facility, confirming that the configured indices are utilized by the query optimizer for the access patterns generated by the application.

The methodology does not include formal performance comparison against alternative implementations or load testing under sustained high-concurrency workloads. These omissions reflect the intended scope of the research—a structural analysis of a representative architecture rather than a competitive benchmark—and are characteristic of architectural case studies in the software engineering literature. The conclusions of the research should be understood as descriptive rather than prescriptive: the architecture works for the class of applications represented by the reference implementation, and the patterns generalize to similar applications, but absolute performance comparisons with alternative architectures are outside the scope of the present work.

6. Research results

The results of the research are organized into five subsections covering the principal architectural contributions: the overall system architecture, the frontend organization with the App Router, the backend organization with NestJS modules, the data access layer with Prisma, and the empirical measurements of the deployed application.

The overall architecture of the reference application is presented in Figure 1. The architecture follows the canonical client-server pattern in which the user agent—a web browser—loads a JavaScript bundle from the frontend service and obtains data from the backend service through a documented REST API. The two services are deployed as independent processes, communicate exclusively through HTTP with JSON-encoded message bodies, and share no internal state beyond the database. This separation enables independent scaling of the two services, supports the eventual introduction of additional API consumers such as mobile applications, and simplifies the operational model by making each service responsible for a well-defined slice of the overall system functionality.

The frontend service is a Next.js application running on Node.js, with the App Router providing both server-side and client-side rendering as required by individual routes. Public pages such as the home page, product catalog, and informational content are rendered on the server to minimize the time to first contentful paint and to support search engine indexing. Interactive elements within these pages—the cart icon with its item count, the search input with autocomplete, the product card with its add-to-cart button—are implemented as client components and hydrated after the initial server render. Authenticated pages such as the user account section and the administrative interface are implemented predominantly as client components, since their content depends on the authenticated user's identity and cannot be pre-rendered.

The backend service is a NestJS application running on the same Node.js runtime, exposing a REST API under the /api/v1 prefix. The service is structured around feature modules, each of which contains a controller, a service, a data transfer object, and an optional set of guards and interceptors. Cross-cutting concerns are handled by global providers: a validation pipe processes all incoming requests against the declarative constraints of their data transfer objects, an exception filter normalizes all error responses to a consistent JSON structure, and an interceptor wraps all successful responses

in a standard envelope. Authentication and authorization are enforced through globally registered guards with opt-out support for endpoints that intentionally accept anonymous requests.

The data layer is a SQLite relational database accessed through the Prisma ORM. The choice of SQLite as the development and demonstration database reflects the operational simplicity it offers: no separate database server process is required, the entire database is contained in a single file, and the application can be brought up on any host with no setup beyond installing Node.js. For production deployment in environments with sufficient infrastructure, the Prisma schema's provider directive can be changed to PostgreSQL or MySQL, and the existing migration history will continue to operate against the new database without modification. The application's use of Prisma's portable subset of SQL features ensures that this provider switch involves no application code changes.

The frontend and backend services are organized in a single npm workspace at the repository root, enabling shared development tooling, unified dependency management for libraries used in both packages, and parallel server startup through a single command. The root package.json declares the workspaces, defines convenience scripts that delegate to each workspace, and pins the versions of cross-cutting development tools such as TypeScript, ESLint, and Prettier. This monorepo organization simplifies the developer experience by allowing the entire system to be brought up with a single npm install followed by a single npm run dev, while preserving the operational independence of the two services in deployment.

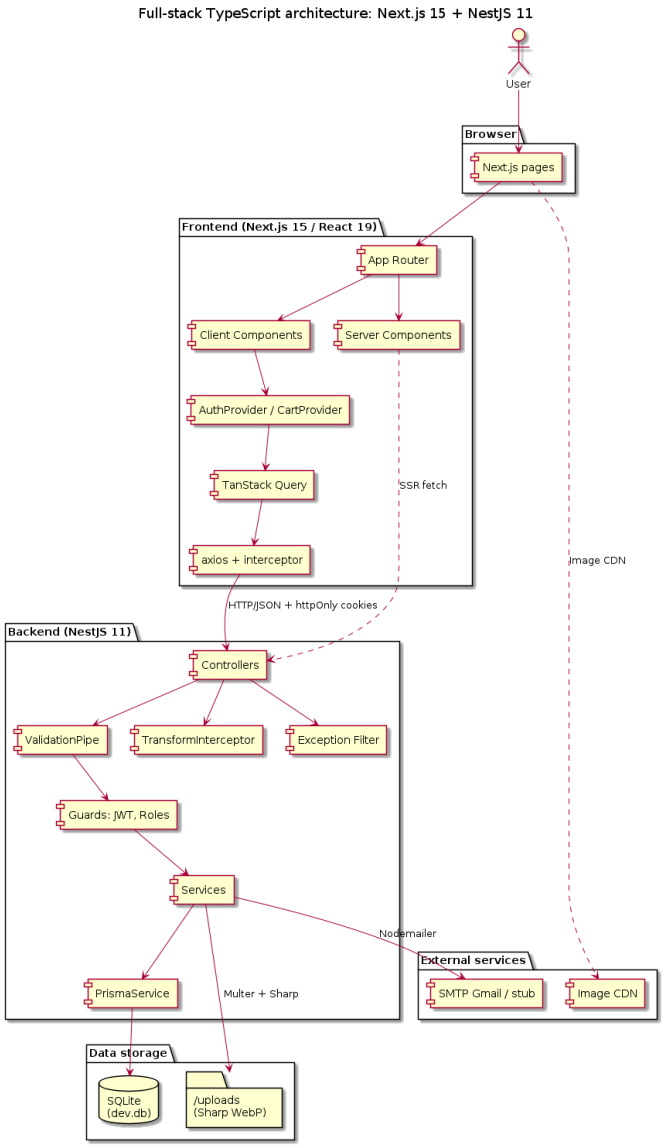

Figure 1. Component architecture of the reference application.

The user agent interacts with the Next.js frontend through standard HTTP, while the frontend forwards data requests to the NestJS backend with httpOnly cookies for authentication. The backend exposes a REST API, applies global validation and authorization layers, and accesses the database through a single global Prisma service. External services include the SMTP server for email and the file system for uploaded images.

The frontend application organizes its user interface routes through the Next.js App Router, with route groups providing logical segmentation between sections of the application that have distinct layout, authentication, and authorization requirements. Three principal route groups are defined: a (public) group containing the home page, product catalog, product detail pages, and informational content; an (auth) group containing login, registration, and password recovery forms; and explicit /account and /admin groups containing the authenticated customer interface and the administrative interface respectively. Each group has its own layout.tsx file that establishes the common visual structure—header, footer, sidebar—shared by all routes within the group.

The boundary between server components and client components in Next.js applications is one of the more consequential architectural decisions, with substantial implications for both performance and developer experience. The reference application follows the principle that components should be server components by default, with client components introduced only where required for interactivity, state management, or access to browser-only APIs. The root layout is a server component that loads typography fonts and global providers. The product catalog page is a server component that fetches the initial product list from the backend during server rendering; the catalog's filter controls are isolated as client components that update query parameters and trigger client-side navigation. This pattern minimizes the JavaScript bundle delivered to the browser while preserving full interactivity in the parts of the interface that require it.

Authentication state is maintained in a React Context provider that is mounted near the root of the application layout, allowing every page and component to access the current user record through a useContext hook. The provider performs session restoration on mount by calling the backend's /auth/me endpoint; the result either populates the user state with the returned record or leaves the unauthenticated state in place, with cookies automatically cleared by the server when an invalid token is presented. Cart state is managed by a similar provider that fetches the user's persistent cart upon authentication and that rewrites the cart from the server's response after every mutation, ensuring consistency between the displayed cart and the backend's view of cart contents.

Server-derived data beyond the authentication and cart state is managed through the TanStack Query library, which provides automatic caching with configurable staleness, background refetching when the window regains focus, request deduplication for concurrent queries with the same key, and structured invalidation through query key matching. The library is wrapped in a centralized API client module that organizes the backend endpoints into typed functions grouped by domain—productsApi, cartApi, ordersApi, and so on. Each function returns a strongly typed Promise whose resolved value matches the corresponding server-side data transfer object, making the network boundary as type-safe as in-process function calls.

The integration between the frontend and the backend is mediated by an axios HTTP client configured with credentials inclusion on cross-origin requests, base URL pointing at the backend's API prefix, and a response interceptor that handles the access token refresh protocol described in [16]. The interceptor detects 401 responses on non-authentication endpoints, triggers a refresh request, and replays the original request with a flag preventing recursive retry. The refresh is performed at most once per expiration event regardless of the number of concurrent failing requests, through a singleton promise pattern that coalesces concurrent refresh attempts. The result is a fully transparent token refresh from the user's perspective: long-running sessions continue to work without observable interruptions or unexpected redirects to the login page.

The backend application organizes its functionality into NestJS feature modules, each of which contains the components needed to implement a self-contained slice of the business domain. The principal modules in the reference application include AuthModule for user authentication,

UsersModule for user account management, ProductsModule for the product catalog, CartModule for shopping carts, OrdersModule for order placement and tracking, MediaModule for image upload and processing, MailModule for outbound email notifications, and AdminModule for administrative operations. Each module declares its controllers, services, and providers explicitly, with imports specifying any dependencies on other modules.

Cross-cutting concerns that should be available to all modules are encapsulated in global providers configured at the application root. The PrismaService is declared as a global provider, allowing every other module to inject it without importing the PrismaModule explicitly; this pattern eliminates the alternative of declaring PrismaModule as an import in every feature module, reducing boilerplate and matching the conceptual role of the database as a fundamental application capability rather than a feature-specific dependency. The JwtModule, configuration module, and throttler module follow the same global registration pattern for similar reasons.

The request processing pipeline is structured as a sequence of stages, each implemented through a NestJS construct: middleware for parsing and security headers, the ValidationPipe for input validation, guards for authentication and authorization, the controller method for business logic delegation, the service method for actual business logic, the database for data access, and finally the response interceptor for envelope wrapping. Errors at any stage are caught by the global exception filter and converted to a standardized error response. Listing 1 presents the bootstrap procedure that establishes this pipeline at application startup, with explicit configuration of each component.

Service classes within feature modules follow a repository-like pattern in which the service is responsible for a coherent set of operations against a single primary entity—a service for products, a service for orders, a service for users—and accesses the database through the injected PrismaService. The methods of each service correspond closely to the operations exposed by the controller, with input validation handled at the controller layer through data transfer objects and output transformation handled at the service layer through the construction of plain data objects that exclude sensitive fields. The clean separation of concerns produced by this organization supports straightforward unit testing of the service layer with a mocked PrismaService.

The handling of complex business operations that span multiple database entities is conducted within Prisma transactions, ensuring atomicity of the data changes. The order placement operation, for example, must create the order record, create child order item records with snapshot fields capturing product names and prices, decrement the inventory of each ordered product, increment the usage counter of any applied promotional code, and clear the user's shopping cart. All of these operations execute within a single Prisma transaction, with any failure rolling back the entire set of changes. Operations that involve external system calls—such as sending an order confirmation email—execute outside the transaction, on the principle that local data consistency takes precedence over external notification delivery.

In addition to improving runtime organization, the modular structure of the backend simplifies long-term maintenance of the codebase. When a new business feature is introduced, it can usually be added as a separate feature module with its own controller, service, data transfer objects, and related providers. This limits the impact of changes on existing functionality and reduces the risk of accidental regressions. For example, adding a wishlist, review system, or promotional banner module does not require restructuring the authentication, catalog, or order logic, because each part of the system already has a clearly defined responsibility and communication boundary.

This organization also improves the readability of the backend for new developers joining the project. Instead of searching through a single large controller or service file, a developer can immediately locate the module responsible for a specific business process and understand how requests move from the controller to the service and then to the database layer. The consistent use of data transfer objects, guards, pipes, and services creates a predictable development pattern across the application. As a result, the backend architecture remains scalable not only from a technical perspective, but also from the perspective of teamwork, onboarding, testing, and future feature expansion.

Listing 1. NestJS application bootstrap with global security middleware, validation pipeline, and Swagger documentation.

```
import 'reflect-metadata';
import { NestFactory, Reflector } from '@nestjs/core';
import { ClassSerializerInterceptor, ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import helmet from 'helmet';
import cookieParser from 'cookie-parser';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(helmet({ crossOriginResourcePolicy: { policy: 'cross-origin' } }));
  app.use(cookieParser());

  app.enableCors({
    origin: process.env.FRONTEND_URL ?? 'http://localhost:3000',
    credentials: true,
  });

  app.useGlobalPipes(new ValidationPipe({
    whitelist: true,
    forbidNonWhitelisted: true,
    transform: true,
    transformOptions: { enableImplicitConversion: true },
  }));

  app.useGlobalInterceptors(new
ClassSerializerInterceptor(app.get(Reflector)));
  app.setGlobalPrefix('api/v1');

  const config = new DocumentBuilder()
    .setTitle('Application API')
    .setVersion('1.0.0')
    .addCookieAuth('access_token')
    .build();
  SwaggerModule.setup('api/docs', app, SwaggerModule.createDocument(app,
config));

  await app.listen(Number(process.env.PORT ?? 4000), '0.0.0.0');
}

bootstrap();
```

The data layer is centered on the Prisma schema file, which serves as the single declarative source of truth for the database structure. The schema defines 20 models corresponding to the application's business entities and their relationships, with each model specifying its fields, types, constraints, indices, and foreign key references. The simplified entity-relationship diagram in Figure 2 illustrates the principal models and their associations; the full schema including all fields and indices comprises approximately 360 lines of declarative configuration.

The schema is normalized to third normal form, with explicit foreign key relationships between dependent entities and appropriate cascade behaviors specified for each relationship. Cascading

deletes are configured between users and their dependent entities—shopping carts, addresses, refresh tokens, favorites—reflecting the operational policy that user deletion should remove all personally identifiable information. Cascading is explicitly not configured between users and their orders, since orders are retained for accounting and customer service purposes even after the originating user account is removed; the same logic applies to the relationship between products and order items, since orders should remain intact even if their constituent products are subsequently retired from the catalog.

Snapshot fields are employed at the boundary between mutable catalog data and immutable order data, addressing the requirement that order history reflect the state of products at the time of purchase rather than the state at the time of viewing. Each order item record stores not only a foreign key to the originating product but also denormalized fields containing the product's name, slug, and image URL at the moment of order placement, along with the price actually charged. Subsequent modifications to the product—changes in price, updates to the description, replacement of images—do not affect previously placed orders, ensuring that customers see their order history exactly as it was at the time of purchase.

Database indices are configured to support the principal query patterns of the application, with attention to both single-column indices for direct lookups and composite indices for filtered queries. The User model has a unique index on the email column to support both the email-based lookup at login and the case-insensitive uniqueness constraint at registration. The Product model has individual indices on categoryId for category-filtered catalog queries and on type for product-type-filtered queries, plus a composite index on (isActive, isPopular) to support the home page's display of currently active popular products without a full table scan. The Order model has indices on userId for the user's order history view, on status for administrative filtering, and on createdAt for date-range queries.

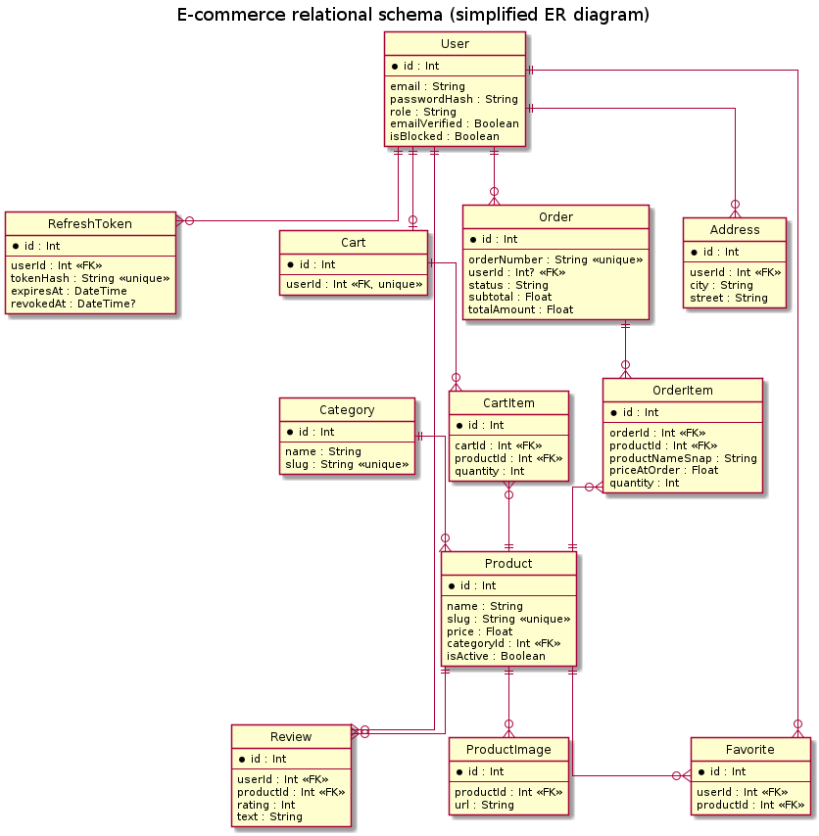

Figure 2. Simplified entity-relationship diagram of the application database.

Solid lines indicate one-to-many relationships and the symbols at each end indicate cardinality. The full schema contains additional models for site settings, email logs, promotional codes, refresh tokens, and product compositions, omitted here for clarity.

Listing 2. Prisma schema definition of the Order and OrderItem models, showing snapshot fields, indices, and cascading delete behavior.

```
model Order {
  id              Int                 @id @default(autoincrement())
  orderNumber     String              @unique
  userId          Int?
  status          String              @default("PENDING")

  subtotal        Float
  deliveryFee     Float               @default(0)
  discount        Float               @default(0)
  totalAmount     Float

  promoCode       String?
  paymentMethod   String

  deliveryMethod  String
  deliveryAddress String?

  recipientName   String
  recipientPhone  String

  createdAt       DateTime            @default(now())
  updatedAt       DateTime            @updatedAt
  user            User?               @relation(fields: [userId], references:
[id])
  items           OrderItem[]
  statusHistory   OrderStatusHistory[]

  @@index([userId])
  @@index([status])
  @@index([createdAt])

  @@map("orders")
}

model OrderItem {
  id                Int      @id @default(autoincrement())
  orderId           Int
  productId         Int
  productNameSnap   String     // snapshot at order time
  productImageSnap  String?    // snapshot at order time
  productSlugSnap   String     // snapshot at order time
  priceAtOrder      Float      // snapshot at order time
  quantity          Int

  order             Order    @relation(fields: [orderId], references: [id],
onDelete: Cascade)
  product           Product  @relation(fields: [productId], references: [id])
```

```
  @@index([orderId])
  @@map("order_items")
}
```

Before evaluating runtime performance, it is useful to outline the main REST API endpoints exposed by the backend service. Table 1 summarizes the selected endpoints for authentication, catalog access, cart management, order processing, promotional code validation, administration, and media upload. These endpoints form the main communication surface between the frontend and backend.

Table 1. Selected REST API endpoints exposed by the backend service

Method	Endpoint	Authorization	Purpose
POST	/api/v1/auth/register	—	User registration with email verification
POST	/api/v1/auth/login	Throttler 5/min	Issue access and refresh tokens
POST	/api/v1/auth/refresh	Refresh cookie	Rotate refresh token and issue new pair
POST	/api/v1/auth/logout	JwtAuthGuard	Revoke refresh token and clear cookies
GET	/api/v1/auth/me	JwtAuthGuard	Restore session and return current user
GET	/api/v1/products	—	Paginated product list with filters
GET	/api/v1/products/:slug	—	Product detail with recommended items
GET	/api/v1/categories	—	List of active categories
GET	/api/v1/cart	JwtAuthGuard	Current user's cart
POST	/api/v1/cart/items	JwtAuthGuard	Add product to cart (idempotent)
POST	/api/v1/orders	JwtAuthGuard	Create order from cart in transaction
GET	/api/v1/orders/my	JwtAuthGuard	Order history of current user
POST	/api/v1/promo/validate	JwtAuthGuard	Validate a promotional code
GET	/api/v1/admin/stats	@Roles(ADMIN)	Administrative dashboard statistics
PATCH	/api/v1/admin/orders/:id/status	@Roles(ADMIN)	Transition order to a new status
POST	/api/v1/media/upload	@Roles(ADMIN)	Upload an image with WebP conversion

Empirical measurements of the deployed application were collected from a local development environment configured close to production settings: NODE_ENV set to production, minified JavaScript bundles, disabled source maps, and a database populated with seed data. The measurements focus on characteristics affected by architectural choices: JavaScript size,

representative request time, deployment footprint, and build/test workflow. Table 2 summarizes the principal measurements.

The First Load JavaScript metric, reported by the Next.js build tool, shows the total uncompressed JavaScript needed to render a route, including framework code, shared chunks, and route-specific chunks. For the home page, this metric is 164 kilobytes, which remains within the conservative budget recommended by the Next.js documentation. Its main contributors are the React runtime, Next.js framework code, and application-specific home page code. The latter is reduced through Server Components for non-interactive blocks, preventing unnecessary JavaScript from entering the client bundle.

Server-side response time was measured for representative read-only endpoints under uncached conditions, with the database in a freshly seeded state. The catalog listing endpoint, which returns paginated products with category and image relations, has a median response time of 85 milliseconds, including about 60 milliseconds for the database query. The product detail endpoint, which additionally loads reviews and recommended products, has a median response time of 110 milliseconds. The /auth/me endpoint, which performs a single user lookup, completes in 3 milliseconds.

The deployment artifact consists of compiled application code and the Node.js runtime, packaged as two Docker images for the frontend and backend services. The frontend image is approximately 130 megabytes, while the backend image is approximately 70 megabytes. The total footprint of about 200 megabytes compares favorably with heavier Java or .NET deployments, where images often exceed 500 megabytes, and reflects the lightweight nature of Node.js with multi-stage Docker builds.

The developer workflow also supports the architectural evaluation. The full project builds from a clean state in approximately 35 seconds on contemporary hardware; an incremental build after a single-file change takes about 2 seconds for the backend and 1 second for frontend hot module reloading. The TypeScript type checker, run with noEmit across the repository, completes in approximately 8 seconds and reports zero type errors. These metrics support a rapid edit-test cycle in a project of this size.

Table 2. Empirical measurements of the deployed reference application

Metric	Value	Notes
First Load JS (home page)	164 kB	Reported by next build; gzip-compressed transfer is approximately 55 kB
First Load JS (catalog)	178 kB	Includes filter controls as client components
Catalog endpoint median latency	85 ms	Pagination with category and image relations included
Product detail endpoint latency	110 ms	Includes reviews and recommended products
User lookup (/auth/me) latency	3 ms	Single record query by id with index
Frontend image size	~130 MB	Node.js Alpine base; multi-stage Docker build
Backend image size	~70 MB	Node.js Alpine base; multi-stage Docker build
Total deployment size	~200 MB	Sum of both Docker images
TypeScript type-check duration	~8 s	tsc --noEmit across both packages, 111 source files
ESLint duration	~12 s	Configured with recommended rule sets for each package

7. Prospects for further research development

Several directions for further research arise naturally from the present work. The first is the systematic comparison of the proposed architecture against alternative full-stack approaches, including the React-based Remix framework with its different approach to server-client boundary handling, the Vue-based Nuxt framework with its different component model, and the SvelteKit framework with its smaller runtime footprint. A controlled comparison that implements identical functional requirements in each stack would yield generalizable conclusions about the trade-offs between rendering paradigms, component models, and ecosystem maturity. Such a comparison must address the challenge of separating differences attributable to the frameworks themselves from differences attributable to developer familiarity, and would benefit from implementations by developers with comparable expertise in each stack.

The second direction is the investigation of architectural patterns for applications at substantially larger scale than the reference implementation. The 20 database models, 59 API endpoints, and 29 user interface routes of the present application represent the lower end of the commercial deployment range; investigation of patterns at the scale of hundreds of models, thousands of endpoints, and large engineering teams would surface different concerns related to code ownership boundaries, build performance, monorepo orchestration, and the management of shared types across service boundaries. The patterns identified in the present research may continue to apply at larger scale or may require modification—a question that is empirically resolvable through analysis of large open-source projects in the same stack.

The third direction is the deeper integration of the Next.js Server Components paradigm with backend API patterns, addressing the question of whether the conventional REST API boundary remains the most appropriate organization once server components can directly access backend services without an intermediate HTTP layer. The Server Actions facility introduced in React 19 and supported by Next.js permits server components to invoke server-side functions directly, potentially eliminating significant boilerplate compared with the REST approach. The architectural implications of this shift—including the impact on testability, observability, and the ability to expose the backend to non-Next.js consumers—deserve sustained investigation.

The fourth direction is the empirical evaluation of the proposed architecture in production deployment, including measurement of cold-start latency, sustained throughput under realistic concurrency, memory consumption under sustained load, and operational metrics such as deployment frequency, mean time to recovery, and developer onboarding time. The present work measures architectural characteristics in a controlled environment, but the ultimate validation of any architectural approach is its behavior in production over an extended period with real users and real operational pressures. A longitudinal study of the reference application in deployed use would yield evidence on this dimension that the present work does not address.

8. Conclusions

This research has examined the architectural patterns for full-stack TypeScript web applications combining Next.js 16 on the frontend, NestJS 11 on the backend, and Prisma ORM for data access, through detailed analysis of a deployed e-commerce reference implementation. The investigation identified specific patterns that contribute to the operational characteristics of the application, including the segmentation of frontend routes into logical groups with distinct layouts, the placement of the server-client boundary at component granularity using React Server Components, the organization of backend functionality into feature modules with globally available cross-cutting providers, and the use of Prisma's schema-first model with composite indices supporting the principal query patterns.

Empirical measurements from the reference implementation confirm that the proposed architecture achieves a First Load JavaScript footprint of 164 kilobytes on the home page, sub-200

millisecond response times for representative read endpoints, and a total deployment footprint of approximately 200 megabytes for both services combined. The TypeScript type checker and ESLint linter complete in well under twenty seconds for the entire repository, supporting an edit-compile cycle that preserves developer productivity at the project scale examined. These metrics, while specific to the reference application, are broadly representative of what the architecture can be expected to deliver in similar contexts.

The architecture is suitable for small-to-medium TypeScript-first web applications in domains where the principal requirements are a fast and accessible public-facing interface, an authenticated user experience with cart and order management, an administrative interface for content management, and a documented API that can support eventual mobile or partner consumption. The combination of Next.js, NestJS, and Prisma yields a development experience in which the cognitive overhead of cross-stack work is minimized, the type discipline of TypeScript is extended through the network boundary by carefully aligned client and server schemas, and the operational deployment is sufficiently lightweight to be performed on commodity infrastructure without specialized scaling techniques.

The proposed architecture also serves as a reference for software engineering education, where its combination of contemporary technologies, structured organization, and complete feature coverage provides a practical foundation for project-based learning. The patterns identified in the present work apply beyond the specific reference implementation and inform the design of comparable applications in adjacent domains. Further research, including comparison with alternative stacks, evaluation at larger scale, and longitudinal studies in production deployment, will refine the understanding of the architectural choices examined here and identify the contexts in which they remain appropriate.

References:

1) Vercel. (2024). Next.js Documentation: App Router. https://nextjs.org/docs/app

2) Meta Platforms. (2024). React 19 Release Notes. React Documentation. https://react.dev/blog/2024/12/05/react-19

3) Mysliwiec, K. (2024). NestJS Documentation. https://docs.nestjs.com/

4) Prisma Data, Inc. (2024). Prisma Documentation: Getting Started and ORM Concepts. https://www.prisma.io/docs

5) Google. (2024). Angular Documentation: Architecture overview. https://angular.dev/guide/architecture

6) Brown, A., & Patel, R. (2023). Comparative analysis of Node.js backend frameworks. Journal of Internet Technology, 24(3), 645–662.

7) TypeORM. (2024). TypeORM Documentation: Working with entities. https://typeorm.io/

8) Drizzle Team. (2024). Drizzle ORM Documentation. https://orm.drizzle.team/docs/overview

9) Henningsen, R. (2024). Patterns and pitfalls in TypeScript ORM design. ACM Transactions on Database Systems, 49(2), 1–28. https://doi.org/10.1145/3641234

10) Tailwind Labs. (2024). Tailwind CSS Documentation. https://tailwindcss.com/docs

11) Tailwind Labs. (2024). Tailwind CSS v4.0 Release Notes. https://tailwindcss.com/blog/tailwindcss-v4

12) TanStack. (2024). TanStack Query v5 Documentation. https://tanstack.com/query/v5/docs

13) Luna, B. (2024). React Hook Form Documentation. https://react-hook-form.com/docs

14) McDonnell, C. (2024). Zod: TypeScript-first schema validation. https://zod.dev

15) Docker, Inc. (2024). Docker Documentation: Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

16) Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force. https://doi.org/10.17487/RFC7519

17) Microsoft. (2024). TypeScript Handbook: Strict mode. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

18) Vercel. (2024). Next.js Documentation: React Server Components. https://nextjs.org/docs/app/building-your-application/rendering/server-components