International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Trustworthy Data-Driven Infrastructure for Digital Philanthropy: A Verification-Centric Platform with Geospatial Discovery and Structured Need–Surplus Matching

Tanishq Chavan
Information technology
Vidyalankar Institute of technology

Deepali Shrikhande
Information technology
Vidyalankar Institute of technology

Aman Singh
Information technology
Vidyalankar Institute of technology

Tanay Shinde
Information technology
Vidyalankar Institute of technology

***

Abstract - Digital philanthropy platforms often separate monetary giving, volunteering, and organization discovery, with uneven verification and accountability. This paper describes Nabhya, a mobile-first platform that verifies NGO registration and eligibility documents before their content becomes public. Public queries also require the organization to remain active, making visibility a data-access rule. NGOs publish quantified wants and offers, while volunteers can receive participation certificates with publicly verifiable codes. The TypeScript monorepo contains an Expo client, a NestJS API with 15 feature modules and 103 endpoints, and a 27-table Drizzle ORM schema on PostgreSQL. Database constraints and row locks enforce uniqueness, event capacity, and fulfillment invariants. An integration suite checks those rules against a PostgreSQL database built from the project's migrations. Two-stage geospatial search and structured WANT/OFFER matching support discovery. Integration, concurrency, and performance tests evaluate the implemented system.

Key Words: NGO verification, digital philanthropy, geospatial discovery, structured matching, in-kind donation, concurrency control.

1. INTRODUCTION

India houses one of the largest non-profit ecosystems in the world but there are two fundamental holes in this ecosystem which prevent citizens from participating. The prospective donor or volunteer has difficulty finding legitimate organizations and understanding their needs in the immediate vicinity. The available systems fill only a portion of this hole. The crowdfunding system provides solutions for raising funds; the volunteers portal compiles volunteer opportunities without being able to authenticate the source; the NGOs directory is the searchable database which does not let you take any further action. The system which can authenticate, contribute and be accountable post-contribution is missing.

In this paper, Nabhya, a mobile-first platform for developing smart social applications using three coupled principles is introduced. Legitimacy, resources and geographic proximity are all determined through queries on the structured data instead of the claims made by the user.

Trust is structured, not advisory. Legal document validation is performed by the platform superadmin on documents like registration certificate, PAN, 80G, and 12A to permit the organization to post anything, and such a filtering process takes place in the data access layer itself, where all public queries are inner-joined with organizations and are filtered based on verification and activation. This makes sure that no public gate will ever be able to disclose any organization which is not verified and activated, regardless of the requirement of the client.

Contribution is two-way and quantifiable, not anecdotal. The organization posts quantified wants (WANT) - ”200 blankets” and quantified offers (OFFER) - ”40 school desks” as transactional accounts.

Recognition is independently verifiable; the code on a participation certificate can be verified by anyone - an employer, college, or any other organization - without any registration or login.

Contributions of this paper to the literature include the following:

1. A model for verification of data access: the client is not responsible for verifying that the organization is genuine; the job of doing this verification is done by the query layer.
2. A relational schema containing 27 tables covering identities, organizations, events, two-way item donation, certificates and accountability, where uniqueness, capacity, and fulfillment constraints are enforced through database constraints and row locking, instead of application level logic.
3. A symmetrical need-donation model, which builds upon the crowdfunding model by connecting organizational needs to publicly accessible surplus.
4. Certificate mechanism without accounts, where each certificate is associated with an event recording its attendance and has a public code verifiable by any third party.
5. Migrations based on correctness; the test database is generated from all committed migrations of the project, therefore invariant violation and schema drift manifest themselves as build errors.

The rest of the paper continues below. Section 2 deals with related work; Section 3 provides the problem definition and design goals; Section 4 describes methodology and technology selection; Section 5 describes system architecture and data model;

Section 6 describes the security model; Section 7 and Section 8 describe testing and evaluation; Section 9 describes limitations; and Section 10 provides the conclusion.

2. RELATED WORK AND RESEARCH MOTIVATION

2.1. Previous Studies

Bekkers and Wiepking analyse the eight motives behind philanthropy, from which awareness of need, solicitation, reputation, and psychological reward imply choices in designing this system [1]. Sargeant and Lee connect trust with relationship commitment in charitable giving in the UK [2]. Other recent research into digital philanthropy highlights the effect of project quality, information, credibility, trust and platform design on online donations [3, 4, 6, 7]. Sander and Zabel study matching donation platforms and show the importance of matching donors' preferences to the chosen nonprofit organization and its projects [5]. These results prompt Nabhya to pay attention to structured information, verification, and need-suitability matching.

Current platforms are specialized rather than integrated. Crowdfunding platforms allow users to donate money to organisations, volunteer platforms match donors with projects, while directories and registers like NGO Darpan [11] verify that the organization is registered but only present it as metadata in search results. Table 1 demonstrates the difference between current platforms and Nabhya's scope.

This comparison reveals that currently existing platforms are predominantly designed for monetary donations, organisation details or volunteering, but are deficient in the other areas. Nabhya provides all of these capabilities along with proximity discovery and structured WANT/OFFER matching, where verification is treated as an access control predicate, and not just an attribute. This approach is well aligned with the recent studies that focus on the importance of trust, structured information and matching in digital donation platforms [3, 4, 5, 6].

2.2. Beyond Concurrency Correctness: Database-Enforced Invariants

Concurrent read then write may fail to apply any updates or allocate more resources than available: event seats, stock or quota. The optimistic concurrency control strategy by Kung and Robinson works well for low-contention situations since any conflict is not detected till the moment of commit, at which point the transaction is rolled back [13]. Nabhya locks rows while accessing them since resolving any conflict may be too expensive in case of high-contention for some limited resources.

Isolation level is also important. Berenson et al. prove that some isolation levels defined by ANSI SQL are inadequate, and that the snapshot isolation level is another multiversion scheme [14]; PostgreSQL's default Read Committed level may allow two concurrent transactions to see the same stale counter value [17]. Without explicit row locking, a read-then-write operation cannot guarantee that it sees the correct value.

Where, if not at the application level, then where should correctness be enforced? Application-level constraints that are dependent on non-serializable behavior of the database cannot be ensured by application checks under concurrent executions, while the persistence layer itself can enforce the same constraint directly [15]. Nabhya enforces uniqueness, capacity and fulfilment rules on database schemas and transactions, and not on the application layer.

Same arguments apply to testing too. Spadini et al. show that developers differentiate between mocking and integration testing because mocks of dependencies often fail to replicate the real database behavior [16]. Integration testing is used to ensure that a unique index or an atomic UPDATE actually exists. Hence, Nabhya performs its integration tests on a real PostgreSQL database generated using the project's committed migrations. What differentiates Nabhya from other systems is that verification is performed using a predicate in the query, not through a label in the profile; item donation is transactional and not just textual; and invariants are verified using the system's migrations and not just a documentation of a possible drifted schema.

3. PROBLEM FORMULATION AND DESIGN OBJECTIVES

3.1. Problem Statement

In this research, a system is designed and implemented which enables verified NGOs to post volunteering positions and material needs and allows the citizen to discover NGOs in the vicinity, volunteer for these organizations, and receive recognition for their volunteering. Verification, moderation, and auditing of the organizations happens via the administrative level, whereas verified organizations get their public interface.

3.2. Design Objectives

There are eight design objectives of the platform.

O1 — Verified Onboarding. Organizations have to provide legal documents which require superadmin approval before they can publish their vacancies, and volunteers should optionally do so with their IDs.

O2 — Concurrency-safe Event Lifecycle. Fully support the lifecycle of the event – creation, publishing, application, reviewing, tracking, and completion – taking into account concurrency of applications.

O3 — Two-Way Item Transfer. Urgent postings of WANTs and OFFERs by organisations; pledges handled transactionally without duplication and over-allocation during concurrent access.

O4 — Verified Discovery. No matter what the client wants to see, directory browsing, feeds and proximity search return only verified and active organisations.

O5 — Engagement and Responsibility. Transparent financial reports are available publicly; bookmarks and reviews are structured and moderated, and so are replies.

O6 — Verified Recognition. Certificates issued only for attendance, are revokable, and are verifiable by anyone without an account.

O7 — Reliability. Administrative modifications are done only through appending; rate-limited for security-sensitive writes; enforcement invariants against a genuine SQL database, not a simulated persistence.

O8 — Securely Settled Financial Payments. The donation amounts are computed by the server itself, and for any settlement of a donation to happen, there must be a webhook that is signed. If the donation has been settled once, then it cannot be settled again.

This integration framework achieves complete coverage of Objective O8. Settlement for payments is tested in Razorpay's testing/sandbox environment, whereas production settlement is not tested because onboarding with merchants is yet to be done.

Table -1: Comparison of selected digital philanthropy platforms and Nabhya.

Platform	Verification	Mon.	In-kind	Vol.	Prox.	Matching
Milaap [8]	Document/KYC	✓	—	—	—	—
GiveIndia [9]	Partner information	✓	—	—	—	—
ConnectFor [10]	Partner NGO	—	—	✓	—	—
NGO Darpan [11]	Registration record	—	—	—	—	—
Nabhya (proposed)	**Query-level gate**	✓	✓	✓	✓	✓

"—" means that the relevant capability was not found in the documented platform features that were examined in this study. "Mon." stands for monetary donations, "Vol." for volunteering, "Prox." for proximity discovery, and "Matching" for structured WANT/OFFER matching.

4. METHODOLOGY

4.1. Development Approach

Nabhya is a pnpm workspace, a Turborepo monorepo written in TypeScript, which consists of a mobile client, a REST API, and shared packages containing the database schema, environment validations, and request/response contracts. Separating client and server into two independently versioned repositories was decided to be undesirable due to the possibility of diverging versions of the contract on both sides.

All requests and responses are Zod schemas stored in a shared package: the server uses them as validators per route, and the client shares the same validator in its form layer using the resolver. This way, the need for a DTO layer, which a separate codebase would require, and a class of runtime errors, which this layer could have been causing, is avoided.

The first prototype was written with a different stack in mind, but the switch to the current stack was made mostly due to the need for a unified codebase for the client, the API, and the validation, which the shared contracts approach requires. Otherwise, there would be two schemas and the need for the same synchronisation that this approach avoids.

4.2. Choice of Technologies

In Table 2, the technology stack and the reasons for choosing them are listed. There are only three technologies that carry the load. Identity is provided by some other party, so the API does not have sessions, hence stays stateless and verifies tokens with the help of remote JWKS. Since there are locking mechanisms in the relational engine, capacity and fulfilment constraints are defined as schema constraints (Section 5), while the integration suite is run against in-process PostgreSQL database which is constructed based on migrations.

5. SYSTEM ARCHITECTURE AND DATA MODEL

5.1. Layered Architecture

Nabhya runs as three cooperating layers. The mobile client (Fig. 1) is an Expo app with route groups organised by role. The API exposes 103 endpoints across 18 controllers in 15 feature modules, backed by 19 services (Fig. 2). The data layer is a 27-table Drizzle ORM schema on Supabase PostgreSQL.

This separation is deliberate: a presentation-layer change doesn't force a schema change, and a business-rule change in the API doesn't force a client rebuild. The shared contract package is the one place both layers deliberately couple.

Screens fetch data through typed `lib/api` services backed by TanStack Query hooks. The authentication state is stored in a Zustand store persisted in the `expo-secure-store`, and Bearer JWTs are refreshed through the Supabase Auth client. Files go straight from the client to Supabase Storage, bypassing the API server.

All incoming requests will hit the `ThrottlerGuard` first:

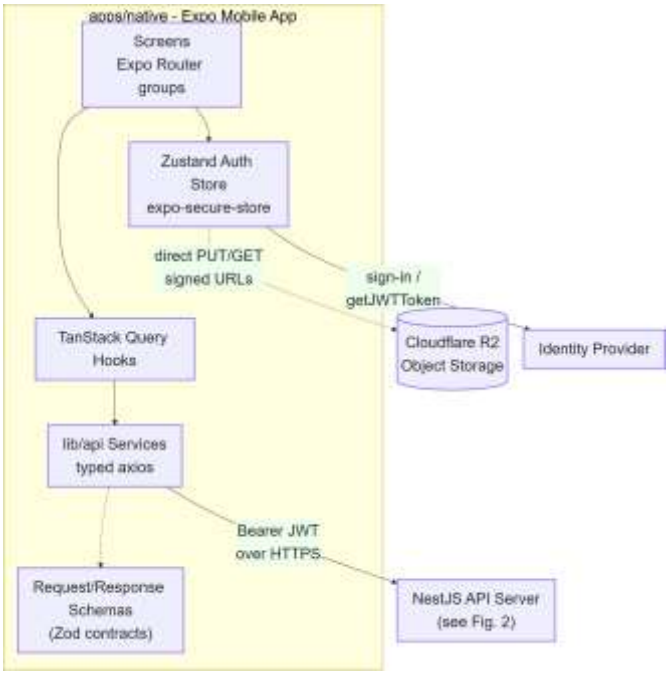


Fig -1: Schematic representation of the mobile client architecture: Expo Router screens, Zustand auth store, TanStack Query hooks and typed API services accessing Supabase Auth for authentication and Supabase Storage for file uploads.

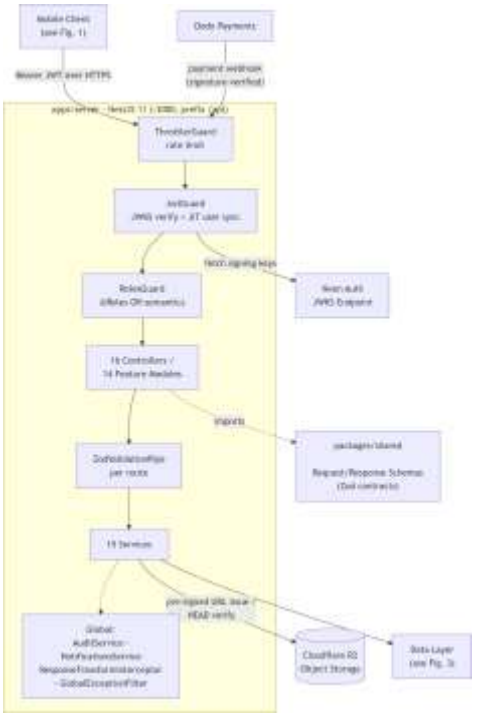


Fig -2: The server side request pipeline with 18 controllers across 15 feature modules and 2 global providers for responding and auditing, plus rate limiting, JWT/JWKS verification, role based authorization, schema validation and more.

Table -2: Technology stack and architectural rationale.

Layer	Technology	Version	Rationale
Mobile client	Expo (React Native), Expo Router	SDK 56	Single codebase for Android and iOS with file-based typed routing
API	NestJS	11	Guards and pipes provide natural insertion points for layered access control
Authentication	Supabase Auth, stateless JWT/JWKS	—	Delegates identity management; API remains stateless via remote signature verification
Data layer	PostgreSQL (Supabase), Drizzle ORM	0.45	Relational constraints and row-level locking with version-controlled typed migrations
Object storage	Supabase Storage, pre-signed URLs	—	File bytes bypass the API, reducing bandwidth and attack surface
Payments	Razorpay, webhook verification	—	Settlement only on cryptographically verified callback
Testing	Vitest, PGlite	4.x / 0.5	In-process PostgreSQL built from production migrations
Runtime	Node.js, pnpm, Turborepo	22 / 10.10	Workspace and task-graph management for the monorepo

generally 120 requests/minute, 10/minute for reviewable requests, checkout for donations, and upload intentions – the payment webhooks and health checks are excluded. After that comes the JwtGuard, which validates the signature against Supabase Auth's JWKS endpoint and syncs the user on demand, after which we have another RolesGuard but this time OR-semantics one. The requests then proceed to one of 18 controllers in 15 modules, each validated with the help of the ZodValidationPipe before being routed to one of 19 services. The payment settlement goes through the same entry point from the other end: Razorpay invokes a signed webhook directly into this pipeline, rather than anything the client can provide.

The data access layer is deliberately kept very simple: services work with PostgreSQL via Drizzle ORM, whereas clients interact with Supabase Storage directly using pre-signed URLS via PUT and GET methods.

5.2. Data Model

The schema holds 27 UUID-keyed tables across eleven schema modules and six functional domains – identity and access, organisations, events, bidirectional item donation, certificates, accountability (Fig. 3). Migrations are committed to the repository and are validated by the integration-test environment. The migration path is therefore exercised as part of testing, while production deployment migration execution is outside the scope of the current evaluation.

Two counters are explicitly denormalized: the accepted_count of an event and the fulfilled_quantity of a need, hence the absence of aggregation per row in public views. Both counters are incremented only in a transaction that first locks the parent document (Section 5.3).

The following three decisions stem from this design. All monetary values are integer minor units, never floats. File storage keys, not URLs, are stored in the files columns, with the ID of the parent object baked into the key for secure downloads without trusting the client. Uniqueness is enforced by the database, not the application: one application per volunteer per event, one review per user per organization, one organization per owner, one certificate per participation, and one donation per provider payment identifier.

5.3. Concurrency Control and Invariant Checking

Two scenarios where there is concurrency of requests competing for a limited resource include capacity and fulfillment accounting. Both scenarios acquire a row level exclusive lock on the parent record prior to reading the counter so that another transaction waits for the former to commit before proceeding with the updated value. Absent of this locking mechanism, in a de-

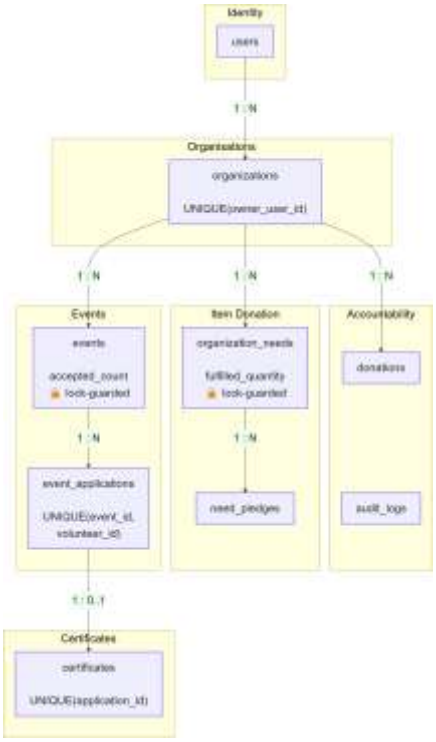


Fig -3: The functional domain grouping of principal entities of the 27-table schema. For legibility purposes Join and Support Tables are not included.

Algorithm 1 Safe capacity-based application acceptance

Require: Application a, parent event e

```
Start a transaction
Acquire FOR UPDATE lock on e
if e.accepted_count ≥ e.volunteers_needed then
    Abort; return HTTP 409 Conflict
end if
a.status ⟶ ACCEPTED
e.accepted_count ⟶ e.accepted_count + 1
Commit a transaction
return HTTP 200
```

fault PostgreSQL isolation mode of READ COMMITTED, two transactions would be able to concurrently read a stale value of the counter and execute [14].

Algorithm 1 outlines the algorithm for event capacity: lock the event, compare the counter with the threshold, and either update it and commit, or roll-back with 409 error if the event is at full capacity. Pledge fulfillment is the same four-step sequence: a pledge locks its parent need, compares the pledged quantity with the capacity left (completely skipping comparison if the need has no definite quantity), then increments the `fulfilled_quantity` counter until it reaches the goal.

5.4. Public Discovery

Every listing publicly exposed in our system always joins organisations table as part of the query itself, rather than via post-processing, and thus suspension of an organisation is automatically reflected in all listings in one fell swoop, with no cascading action per entity necessary.

The geospatial discovery process executes a bounding box query over the lat/long grid in order to obtain candidates, and calculates the Haversine distance over them: city-level spatial proximity without requiring any dedicated geospatial extension. The same layer enables organisational need (WANT) discovery based on public surplus (OFFER) in category, quantity, urgency and location.

5.5. Payment Settlement

Initially, a donation is in a pending state where the client just indicates the organization it wishes to donate to, along with the amount to be donated. The server always computes and logs the actual settlement amount. The server creates an order with Razorpay and gives an order ID and public key to the client who then loads the Razorpay native in-app payment page. Only a Razorpay webhook having a valid signature can close a donation as complete. There is a uniqueness constraint enforced on the provider payment identifier, and hence it becomes impossible for any repeated webhook call to double charge a donation, as tested in Section 8.4.

6. SECURITY MODEL

6.1. Threat Model

It is assumed that the adversary can fully control the client side and manipulate the mobile application in any way possible - modification, decompilation and even use direct HTTP requests to circumvent the client side.These parties are considered trustworthy since the cryptographically signed assertions issued by them – JWTs, signed pre-signed URL, webhook signatures – are considered non-falsifiable, and the database is considered reliable. Protecting against denial-of-service attacks other than client-based rate-limiting is not within the scope of this project, neither is transport-level encryption, covered by TLS.

6.2. Authentication and Authorisation

Denial is the default state of the system: all endpoints have to be authenticated unless declared public. Every request must go through a fixed sequence of steps – rate-limiting, JWT validation, role-based authorisation and schema validation – before reaching business logic, which takes place in guards, and not in route handlers, so that no new route can bypass any of these steps.

The credentials are delegated to an external provider and are not handled by the API. The request is made with a bearer token, which will be validated with the provider's remote JWKS endpoint, leaving the API stateless. The first successful presentation will sync the user record just-in-time with first checking for the provider's subject id and fallback to email for existing account. The account state (whether it is suspended or deleted) is not part of JWT validity period – it will be checked on each call at this guard point, not just at sign-on, making revocation effective immediately.

Each endpoint either needs to be explicitly marked as public or needs authentication. At registration, there is no role assignment from another source, but a role-gated action will not be available until it is declared; roles checks work with OR-semantics against declared roles.

User identity asserted by the client will never be trusted: tokens will have the caller's identity and affiliation, so it is not possible for a user to assume the identity of another user or pretend to work for any organisation other than their own. Endpoints that scope themselves to an organisation read the organisation associated with the token only, not any request parameter. This eliminates an entire class of privilege escalation because there is no identifier parameter to substitute anymore. If an endpoint publishes public content, another condition is added: the resolved organisation should be verified, too.

Such an approach is compared to the integration testing framework described in Section 7. It does not mock the persistence layer and its semantics but stresses volunteer capacity, fulfilment calculation, verification checks, constraints, and transactions using real PostgreSQL behavior.

6.3. Verify is an Access Control Predicate

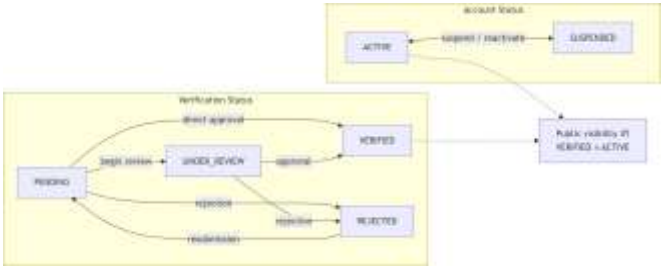


Fig -4: The lifecycle of verification for an organisation. Verification and account status are separate concepts; public availability is their intersection.

Organisational verification proceeds according to the lifecycle shown in Fig. 4: registration certificate, PAN, 80G, 12A, GST, and bank proof are checked at the organisational level; a superadmin examines the documents and makes the final decision about the whole organisation. Here, the verification process denotes that the organisational documents that have been submitted are reviewed, and the organisation is marked as verified in the database. Thus, it becomes a document-based eligibil-

ity check for public visibility and not an external audit of the programmes, financial records, or impact assessment of the organisation's beneficiaries.

The above predicate is calculated within the same query that constructs the public listing in question and is not filtered out from the results in any way. Suspend the organisation and its entire public surface will immediately be updated the next time it is queried as long as the shared query helpers are used by all endpoints.

6.4. Limitations of the Guarantee of Verification

It is assumed here that the submitted documents satisfy the requirements for an Indian non-profit entity registration and were judged by a human as believable; it is neither an audit nor an approval of an organisation's credibility. As it stands, documents have not been compared to any official registry yet, so a well-forged document may go through. The verification is done upon review and is not done periodically, although some certificates like 80G expire over time. Program effectiveness, financial propriety, and beneficiary impact are not evaluated. This guarantee therefore is about the eligibility for the visibility in terms of reviewed documents, not a broader organisational effectiveness and credibility.

Cross-checking program documents against the national NGO registry and the foreign contribution register would make the guarantee more robust; see Section 9.

6.5. Data Handling

There are no file byte transfers through the API. When the client requests a file upload, it gets a temporary pre-signed URL as well as an object key containing the entity's identifier and uploads to storage using this URL. The object existence and its size are then verified via a POST request. The reads are done similarly but in reverse order – a signed URL is provided after verification of the ownership of the document via its key.

The verification documents and the ID documents of the volunteers remain confidential in contrast to the financial transparency documents created by the organization and published. All the monetary values are in integer minor units.

Any administrative actions are logged in an append-only fashion: the action, its actor and entity, the before and after values and the originating address are all recorded. This log entry doesn't block the actual action that takes place as it is being logged because an audit error shouldn't prevent a valid action from taking place.

6.6. Payment Integrity

The donation amount calculation happens on the server side only and cannot be manipulated by the client. The donation can only be completed through a webhook from a signature-checked provider and not through a client's request. There is a unique constraint placed on the provider payment identifier, which ensures that the same payment cannot be settled through multiple webhooks sent to the server; this has been tested in Section 8.4.

7. TEST METHODOLOGY

7.1. Integration Tests Against an Actual Database Engine

The integration suite is run using Vitest on an actual PostgreSQL engine for testing purposes. The database used in testing is PGlite in-process database which has been generated using the committed migrations of the project instead of the ORM schema definition so that the tests performed are actually against PostgreSQL's behaviour.

All of the migration suites to be used for testing were provided directly by the project itself. According to Spadini et al., a mock collaborator by itself will not be enough to verify a properly integrated module [16]. For example, in case when the behaviour depends upon a unique index or a row-level lock, a mocked persistence layer will ensure the validity of control flow, but the semantics of the database operations will remain untested.

Our testing suite includes 16 specifications of API, a harness to build the database and the service graphs, row factories and a throw-away configuration of the environment, without requiring any credentials of developers for CI. The functional invariant tests are done sequentially per each invariant, while the contention testing is done separately as described in Section 8.3 for Algorithm 1.

7.2. Invariant Coverage

The tests are organised in terms of invariant types rather than paths of execution (Section 5), where Table 3 shows the relationship between invariant type, enforcement mechanism, and an example of test case that triggers the invariant.

7.3. Build Time Dependency Verification

Another phase verifies all of the dependencies of controllers and services. Esbuild is used here, and a dependency that does not explicitly have an injection token cannot be verified; it fails at runtime. It will resolve to `undefined` at the point where it occurs, but verifying its dependency at build time would make it a compilation issue.

8. EVALUATION AND DISCUSSION

8.1. Invariant Verification

In Table 3, all six invariant types pass through the integration suite. This works because the test database builds using every migration file committed up until that point, thus verifying these invariants based on the actual deployment database schema, and not some separate document of it.

This verification method has actual teeth: a deployment with migrations that define a constraint will always deploy with said constraint, and it will fail if any code path violates the constraint. There is no such thing for application code because invariants in application logic apply to those code paths.

8.2. Quantitative Performance Evaluation

As part of the functional integration tests, quantitative testing of the API and database workloads was done. The test assesses the latency and throughput of the workloads.

These measurement results capture the performance characteristics of the existing system under the particular workload being tested.

8.3. Concurrency Evaluation

In addition to evaluating the capacity invariant on a single request, we also evaluated this invariant under multiple concurrent requests. Several requests were fired at the same time for an event having a fixed number of volunteers' capacity. This test measured the successful and rejected requests and verified whether the number of accepted requests surpassed the specified capacity.

Within all levels of contention considered, the number of accepted applications stayed bounded within the configured size of

Table -3: Invariant types, enforcement mechanism, and representative test case.

Invariant type	Enforcement	Representative case
Uniqueness	UNIQUE(event_id, volunteer_id)	Duplicate application receives HTTP 409
Capacity accounting	Row lock on parent event	Acceptance of applications exceeding capacity is declined
Fulfilment accounting	Row lock on parent need	Pledge completion increments counter atomically
Verification gating	Predicate at query level	Unverified organisations are not listed and cannot be found in proximity search
Attendance-gated issuance	UNIQUE(application_id) and attendance condition	Certificate issuance denied without attendance
Payment replay safety	Unique payment ID by provider	Duplicate payment hook settlement fails

Table -4: Results of Quantitative Performance Testing.

Workload	P50 (ms)	P95 (ms)	P99 (ms)	Throughput (req/s)
Public Discovery	42	78	105	118
Proximity Search	51	94	127	102
WANT/OFFER matching	63	116	154	86
Event Application	37	71	98	132

the event, with no over-allocation detected. This offers empirical evidence of row locking mechanism outlined in Algorithm 1, while higher contention and production-grade loads have been out of the scope of the experiment.

8.4. Gating Verification and Payment Replay Safety

We checked two properties directly: verification gating and payment replay safety.

The former was checked by generating organisations in each state of verification and accounts, and querying all available endpoints. An organisation which is not simultaneously verified and active would not be present neither in the directory, nor in event feed, nor needs feed or proximity search; as the predicate is included in the query itself, but not in the response filter, it applies to any endpoint which uses shared query helper, even newly added after the experiment.

In order to ensure that a replay is safe, two provider payment identifiers were provided twice through duplicate webhooks with valid signatures. The unique constraint prevented the second delivery, thus resulting in only one settlement of the donation. This payment path is completely covered by the suite; its deployment in production is yet to be done since the merchant verification by Razorpay is pending, hence it is running in the test environment of the provider.

8.5. Discussion

Collectively, the results on functionality, concurrency, and quantitativeness give proof to the technical properties being achieved by Nabhya. Query-level verification ensures that unverified or non-active organizations do not show up on the evaluated public paths, and the database constraints and row-level locking ensure that the tested integrity invariants are being enforced. The concurrency testing tests the capacity invariant with simultaneous requests, and the performance testing gives the baseline quantitative results for representative workloads.

9. LIMITATIONS AND FUTURE SCOPE

9.1. Current Implementation Limitations

A web console hasn't been created yet; all administration and super-administration actions can be done via mobile app exclusively, even though reviews are best done on the desktop.

Push notifications can only be received inside the application; there is no notification either by e-mail or push about the decision; a volunteer will only find out about the decision when opening the application.

Pagination is provided on every list resource endpoint, but only the first page from every list feed is consumed by the client. For proximity search, the general purpose bounding box combined with the Haversine distance calculation is used instead of the specialized geospatial index; this works well enough on city level, but could be revisited for larger deployments.

In non-production environments, the schema is being applied at boot time directly via synchronization instead of migrations; only the testing suite tests the migration process while the application start-up sequence needs to run migrations first.

A client-side route guard is always optional because authentication happens on the API side, and the client merely looks up the role stored on its end to determine what to display, meaning that if the role is stale, it will show the incorrect navigation group briefly.

Monetary donations are completely integrated end to end. Production deployment will have to wait for the completion of the onboarding process on Razorpay's merchant identity verification system.

9.2. Limitations of the Evaluation

Functional invariant testing in Section 8.1 runs sequentially for each invariant, and concurrency testing occurs separately in Section 8.3. Currently, concurrency testing occurs in a limited setup and does not validate behavior at production-scale contention levels.

Field testing with real donors, volunteers, and NGOs has not yet occurred. Thus, the evaluation is unable to determine adoption rates, sustained usage, the realistic workload associated with the verification queue, or if users find their implementation of these mechanisms makes them more trusting.

9.3. Future Work

Further scalability and concurrency testing using higher request counts and distributed systems is still future work, based upon the results from contention in Section 8.3. In addition to this, there is another potential future work line based on the audit log and pledge history data, which can be used to detect unusual organisation behavior through anomaly detection. Another area that needs automation is the verification queue, as described in

Table -5: Concurrent capacity allocation results.

Concurrent requests	Capacity	Accepted	Over-allocation
10	5	5	0
25	10	10	0
50	20	20	0

Section 6.4.

From the product point of view: a web-based admin interface, email and push notifications, client feed pagination, and the right geospatial index when proximity searches are scaled up. Analytics regarding organizational performance (app funnel, donation conversion rate, donations trends) would enable organizations to see how well they are doing the same way as the superadmin does now, and localization of languages besides English is crucial in India, not merely optional.

The most important enhancement for the verification process is to compare the information against the National Register [11] and the foreign contribution register [12], so that the problem described in Section 6.4 will be a mere verification of information in registers available to the authorizing agency. Re-verification will ensure that the system is able to identify expired 80G certificates and other such instruments.

9.4. Governance Considerations

The technical aspects of production deployment have other issues associated with them. The scope of the verification process needs to be understood by its users: a verification mechanism that is trusted to an extent it shouldn't be serves all those using it poorly. Publishing media and information on beneficiaries must observe the minimum necessary disclosure principle based on consent as well as the fact that those depicted are beneficiaries and not the organization. A deployed solution would require a moderation policy, grievance handling policy, and data retention policy as well; the system stores personal identification data and financial information, and the time to retain it is a governance issue addressed in the architecture but not answered.

10. CONCLUSION

Nabhya was presented in this paper, and the structural features it has – organizational trust, item level bidirectional giving, and volunteer verification – are built into the platform from the very beginning.

Three architectural properties can be demonstrated using the evaluation results. First, unverified or suspended entities are prevented from appearing in the evaluated public discovery mechanisms by predicates on the query level. Second, the 27-table schema uses database constraints and transaction locks to enforce the properties of uniqueness, capacity, and fulfillment. Third, integration and concurrency testing verifies these properties on a PostgreSQL database produced by migration, while quantitative evaluation provides performance baselines.

Deployment in the field with actual donors, volunteers, and NGOs, along with longitudinal evaluation of adoption and perceived trustworthiness, is a task for the future. Improving verification by performing regular cross-validation and re-verification against external registries is also required in production use (Section 6.4). With all these limitations, Nabhya demonstrates a testable blueprint of trustworthy infrastructure for data-driven smart social applications.

REFERENCES

1. R. Bekkers and P. Wiepking, "A literature review of empirical studies of philanthropy: Eight mechanisms that drive charitable giving," *Nonprofit and Voluntary Sector Quarterly*, vol. 40, no. 5, pp. 924–973, 2011.
2. A. Sargeant and S. Lee, "Donor trust and relationship commitment in the U.K. charity sector: The impact on behavior," *Nonprofit and Voluntary Sector Quarterly*, vol. 33, no. 2, pp. 185–202, 2004.
3. C. van Teunenbroek, C. Dalla Chiesa, and L. Hesse, "The contribution of crowdfunding for philanthropy: A systematic review and framework of donation and reward crowdfunding," *Journal of Philanthropy and Marketing*, vol. 28, no. 3, e1791, 2023. doi:10.1002/nvsm.1791.
4. A. Prashar and P. Gupta, "How to build trust in Gen Y in online donation crowdfunding: An experimental study," *Behaviour & Information Technology*, vol. 43, no. 4, pp. 677–694, 2024. doi:10.1080/0144929X.2023.2183061.
5. P. Sander and J. Zabel, "It is a match! How donors and nonprofit organizations come together on a matching donation platform," *Journal of Philanthropy and Marketing*, vol. 29, no. 2, e1845, 2024. doi:10.1002/nvsm.1845.
6. V. Bhatt, S. Pandey, R. Patel, S. Ghodake, H. Jariwala, and S. Thomas, "Predicting online donation intention in donation-based crowdfunding apps: A multi-stage SEM-ANN-NCA model integrating anthropomorphism, satisfaction, trust, and privacy concerns," *Journal of Nonprofit & Public Sector Marketing*, vol. 37, no. 1, pp. 80–104, 2025. doi:10.1080/10495142.2024.2351012.
7. Y. Rimawati, Hamidah, and Ardianto, "Digital users perceptions of philanthropic donations across social media and institutional websites," *Journal of Philanthropy and Marketing*, vol. 31, no. 1, e70046, 2026. doi:10.1002/nvsm.70046.
8. Milaap, "Trust & Verification Policy." [Online]. Available: https://crowdfunding.milaap.org/milaap-trust-verification-policy/
9. GiveIndia, "About GiveIndia." [Online]. Available: https://ngo.giveindia.org/
10. ConnectFor, "Volunteering Platform: Connecting Volunteers to NGOs." [Online]. Available: https://www.connectfor.org/
11. Government of India, "NGO Darpan Portal." [Online]. Available: https://ngodarpan.gov.in/
12. Ministry of Home Affairs, Government of India, "Foreign Contribution Regulation Act services." [Online]. Available: https://fcraonline.nic.in/
13. H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, Jun. 1981.
14. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. 1995 ACM SIGMOD Int. Conf. Management of Data*, San Jose, CA, USA, 1995, pp. 1–10.
15. P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in *Proc. 2015 ACM SIGMOD Int. Conf. Management of Data*, Melbourne, VIC, Australia, 2015, pp. 1327–1342.
16. D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? An empirical study on mocking practices," in *Proc. IEEE/ACM 14th Int. Conf. Mining Software Repositories (MSR)*, Buenos Aires, Argentina, 2017, pp. 402–412.
17. PostgreSQL Global Development Group, "PostgreSQL documentation." [Online]. Available: https://www.postgresql.org/docs/