

# Evaluation of a Verifiable DBMS (Database Management System)

Ivan Ribeiro  
*Master's Student*  
University of Minho  
Braga, Portugal  
pg55950@uminho.pt

Francisco Ferreira  
*Master's Student*  
University of Minho  
Braga, Portugal  
pg55942@uminho.pt

Diogo Marques  
*Master's Student*  
University of Minho  
Braga, Portugal  
pg55931@uminho.pt

Pedro Carvalho  
*Master's Student*  
University of Minho  
Braga, Portugal  
pg55997@uminho.pt

**Abstract**—Verifiable database management systems (verifiable DBMSs) aim to provide cryptographic guarantees about the correctness of query answers, allowing users to independently verify the integrity of the underlying data and computations. The PeTall project, developed in collaboration with INCM and awarded under the IN3+ innovation initiative, proposes a set of components that enable such verifiability through zero-knowledge techniques. This work presents an experimental evaluation of these components by integrating them into a prototype application designed for an energy community scenario. Our goal is to assess the suitability, limitations, and practical usability of the PeTall verifiable DBMS when applied to real-world application development. Through this case study, we analyze system behavior, data immutability constraints, adversarial considerations, and user interaction requirements. The results highlight both the potential and the challenges of adopting verifiable DBMS technologies, offering insights into their maturity and applicability in practice.

**Index Terms**—Verifiable DBMS, Zero-Knowledge Proofs, PeTall, Energy Communities

## I. INTRODUCTION

Ensuring correctness and integrity of data returned from a database is a fundamental requirement in applications whose trust, transparency, and auditability are crucial. Although traditional DBMSs support trusted administrators and secure infrastructures, they cannot cryptographically prove the correctness of query answers. Verifiable DBMSs aim to bridge this gap by enabling independent validation from users over query results, reducing the amount of trust required in system operators.

The PeTall project, developed in cooperation between research institutions and INCM, and distinguished with an IN3+ innovation award, investigates this paradigm through the development of components that enable support for query verifiability using zero-knowledge mechanisms. Instead of implementing a DBMS from scratch, the work that follows experimentally evaluates the PeTall system and assesses its readiness for real-world application development.

To structure this assessment, we define and implement a prototype using a relevant use case: an energy community, where several actors both contribute and consume energy, and require transparent distribution rules on the basis of immutable data. This setting provides realistic challenges for verifiable DBMS technology regarding adversarial behavior, fairness constraints, and the need for high-quality user interaction.

## II. CASE STUDY

Residential adoption of solar photovoltaic systems has increased significantly over the past few years [1]. Although these households frequently produce surplus energy, a substantial portion of this energy is wasted due to limitations in storage or local consumption. Energy Communities address this inefficiency by enabling households to share excess production with other members, improving sustainability and reducing collective energy costs.

However, sharing energy fairly within a community is non-trivial. Different households exhibit distinct production and consumption patterns, and a naive distribution mechanism may favor certain members disproportionately. For instance, users who consistently produce more energy could repeatedly contribute more than they receive, resulting in systematic unfairness. The energy distribution must also be provable; otherwise, the energy providers themselves might fabricate or omit data to the user for their own financial gain, possibly controlling when energy is shared or sold. These fairness concerns motivate the need for transparent and auditable rules governing energy allocation.

### A. Problem

One straightforward solution to guaranteeing fairness is to make all production and consumption records publicly available. If every household can inspect every other household's energy history, then each member can verify whether distributions were performed correctly and whether they have received an equitable share.

While an effective solution, this approach introduces a new problem regarding the privacy of the users. Energy usage patterns may reveal highly sensitive information about the occupancy status of the household, such as when residents are home, asleep, at work, or away on vacation, or the type of devices that are being powered. Such information could be exploited by malicious actors, including burglars or other adversaries seeking to infer schedules, household routines, profile households based on estimated wealth or asset levels, or any other sensitive information. Therefore, any realistic system must balance auditability with strong privacy guarantees.

### B. Zero-Knowledge as a Solution

To reconcile fairness with privacy, the case study relies on zero-knowledge proofs (ZKPs). PeTall's verifiable DBMS produces cryptographic proofs over immutable snapshots of the database. These proofs allow each participant to verify the correctness of the computed energy distribution without learning the individual production or consumption values of other households.

For example, if a user contributed 10 kWh to the community, the system can produce a proof that they are entitled to receive at least 10 kWh at a later time, while not revealing to this user how much energy any other user contributed or consumed.

### C. Energy Distribution Rule

As mentioned above, energy sharing among community members must follow rules that minimize differences between consumption and production, but there are several ways to achieve this goal. With this in mind, each community has a calculation rule, defined at the time of its creation, from which the coefficients that dictate the allocation of energy to each user will be calculated.

It should be mentioned that this rule must necessarily be immutable, otherwise different logics would be applied in snapshots of the same energy community, which would not make any sense, as we would try to compensate through formula X, energy that was consumed according to formula Y.

## III. APPLICATION PARTS

### A. Database

The application uses PostgreSQL 16 [2], deployed via Docker [3] for simplified setup and portability.

#### a) Schema:

With this requirements in mind we decided on a final database schema which can be seen in Section VI.A.

- 1) **User** Table I: stores registered users in the system with each having an id (unique), name, email and a isAdmin flag which determines if the user has admin privileges or not.
- 2) **Key** Table II: manages user authentication credentials, allowing multiple auth provides like email and google. With this table we can allow a user to have multiple login methods associated with the same account.

- 3) **Session** Table III: maintains active user sessions, each having an expiration date and a unique identifier allowing us to manage user sessions when necessary.
- 4) **Community** Table IV: represents the energy communities in the system.
- 5) **Community-user** Table V: records which users belong to which communities, allowing users to belong to multiple communities and multiple users within a community.
- 6) **Community-manager** Table VI: defines which users are managers of which communities.
- 7) **Energy record** Table VII: is the primary table for energy distribution calculations and verification using zero-knowledge proofs. It stores energy records for each user in each community. Each record contains the users generated and consume energy (kWh), and consumer and seller prices. A new record is generated each 15 minutes to simulate how an electric meter should work.

### B. Community Backend

This component of the application was built with Rust [4] and Axum [5], using the *sqlx* crate [6]. It exposes endpoints that allow the frontend to retrieve and update data from the database. For example:

- GET /community/{id}: retrieves information on a specific community, like its name and image.
- POST /community/{id}/energy: retrieves energy records given a specific community and user. The user is extracted from the JWT, ensuring only a properly authenticated user can access his, and only his, data records.
- POST /community/{id}/stats: given a community and authenticated user like before, retrieves aggregated data over some period of time. For this purpose, it also receives a stats filter, specifying the desired start and end times, and the granularity of the data. This allows leveraging the database itself for aggregating data, instead of retrieving large batches of records to the frontend.
- Auth endpoints at /auth/...: provide an API for users to register, login, and interface with OAuth.

To improve our proof-of-concept functionality, two automatic seeding features were added:

- 1) When a user is added to a community, fake energy records using reasonable values are added up to 90 days before the current date. This ensures the frontend can immediately show data and statistics that need to aggregate data over some time period.
- 2) Every 15 minutes, a new fake energy record is added to each user, for each community that he belongs to. This allows simulating real conditions, where data is constantly being fed to the applications as the electricity meter periodically produces records.

### C. Community Frontend

We have used Svelte [7], Tailwind [8] and shadcn [9] to build our frontend. This allowed the user to see energy records in a pleasant and useful manner and also allows admins to create and manage communities.

- The user logs in the application inserting the username and password or logging into Google OAuth by sending a POST request to the backend with the credentials to the `/api/auth/login` endpoint
- The backend validates the credentials and answers with a `session ID` that is stored as a cookie which will be used in all subsequent requests to authenticated endpoints such as `/community/{id}`.
- The frontend displays aggregated statistics about energy consumption and generation for each of the user's communities. The data comes pre-aggregated from the backend depending on the granularity the users chooses. For instance, if the user wants to see energy consumed and generated each month, the backend will group the data depending on its month and send it to the frontend, improving the system's performance and thus usability.

The user can also click on a validate button for each energy record. This functionality will be described in the next component.

### D. Trusted-Entity

To ensure that the zero-knowledge internals are operated by an independent and trusted party—such as, for example, Imprensa Nacional-Casa da Moeda—we delegate these responsibilities to a dedicated Trusted-Entity service. This service is solely responsible for validating energy records by communicating with the Zero-Knowledge (ZK) service, and shields the Community Backend from having direct access to or control over the ZK internals. This service was implemented in SvelteKit [7].

As shown in sequence diagram Fig. 3, when a community user clicks “Validate”, the community frontend asks the Community Backend to mint a short-lived JWT via `GET /sign-energy-record-validation/{record}`. That token carries the user ID (taken from the user's authenticated session) plus the record ID, and it is signed with a private key known exclusively to the Community Backend; the matching public key is known to the Trusted-Entity service.

The frontend then calls the Trusted-Entity `POST /validate`, passing the JWT. The Trusted-Entity verifies the signature and expiry, extracts the user and record identifiers, and requests a ZK proof for that record. Once the ZK service returns the proof, the Trusted-Entity stores both the proof and record metadata in a user-scoped session (keyed by record ID) and issues a session cookie.

When the frontend later loads the validation page via `GET /validate?query={recordId}`, the Trusted-Entity reads the session cookie, retrieves the stored proof and record data, and renders the validation UI. Because the Community Backend only signs tokens for authenticated users and for records they own, no user can trigger validation for someone else's data. The session also allows that the user cannot share the proof with other users.

### E. Zero-Knowledge Component

Because the production Zero-Knowledge database was not ready during this assignment, we created a proof-of-concept service in Rust [4] and Axum [5] that mimics the responsibilities of the future ZK database. In the real deployment, this component would ingest the Community Backend's encrypted daily snapshots, and produce proofs over these snapshots.

For the POC, the ZK component exposes a single endpoint (`GET /validate`) that, when triggered by the Trusted-Entity service, fetches the requested energy record from the Community Backend (using a temporary endpoint `GET /energy-record-unauthenticated/{record}`) and returns both the raw values and a randomized string that stands in for the eventual zero-knowledge proof. This keeps the Trusted-Entity interface stable while allowing us to focus on the rest of the validation workflow.

## IV. INTERACTIONS BETWEEN COMPONENTS

To make the system truly functional, the various components must communicate with each other and share information, with the following flows being particularly noteworthy:

### A. Register User

As shown in sequence diagram Fig. 1, the user submits a registration form, sending a `POST /auth/register` request with username, email, and password in the body. The system creates the new user if it doesn't exist, generates and saves a `session ID`. Finally, the browser stores the `session ID`, automatically logging the user in. The entire process creates an account and authenticates the user in one flow.

### B. Remove User from Community

In the sequence shown in Fig. 2, the admin selects “remove user”, triggering a `DELETE` request to `/admin/community/{community.id}/user`. The request is processed through the system, which checks the access permissions and deletes the `userCommunity` record from the database. Once confirmed, an `No Content` propagates back, and the UI updates to reflect that the user has been removed from the community.

### C. Validate energy record

Fig. 3 already described in Section III.D.

## V. KEY FINDINGS

Overall, the implementation demonstrates that the PeTall components can be assembled into a coherent and working validation flow, giving us confidence that the architecture moves into a functioning prototype, and the zero-knowledge is applicable in the real world.

The JWT-based handshake proved particularly effective: it keeps the Community Backend and the Trusted-Entity synchronized without duplicating authentication logic, while the signed tokens create a secure point-to-point channel that resists tampering even if the components are hosted by different organizations.

Through this exercise we also found that the energy records need to be immutable. Because proofs are generated from that history and are re-validated at any time, records cannot be removed or rewritten; only softer relationships (such as user-to-community memberships) that no validation is performed on them may evolve independently without invalidating proof material.

Finally, once the zero-knowledge functionality is fully integrated, there will inevitably be a delay between when a new energy record is created and when it becomes eligible for validation. This results from the need to synchronize data between the community database and the zero-knowledge database, which are separate systems. One practical solution is to periodically replicate data, for example through daily snapshots. Accordingly, the application should account for this lag and clearly communicate to users when new records will become available for verification.

## VI. CONCLUSION

This project successfully delivered an architecture suited for a real-world application. Choosing Rust for the backend supplied a strongly typed, fail-safe, and fast runtime that can scale to large user loads. The surrounding ecosystem, Axum, SQLx, and other crates further streamlined development, making the development an enjoyable experience. On the frontend, Svelte combined with Tailwind and shadcn provided a consistent, user-friendly UI built from reusable components, and working in Svelte especially pleasant throughout the project.

Regular meetings with the professors were instrumental for iterative refinement, ensuring the solution evolved alongside new insights. While the problem initially appeared straightforward, deeper investigation revealed non-trivial challenges that demanded deliberate solutions.

Future work includes integrating the production Zero-Knowledge service with data replication and, in practice, energy communities rely on distinct algorithms to distribute excess energy, so supporting provider-specific proof schemes will be essential.

## REFERENCES

- [1] [Online]. Available: <https://www.statista.com/topics/13626/residential-solar-photovoltaics-in-europe/>
- [2] "PostgreSQL 16." [Online]. Available: <https://www.postgresql.org/docs/16/>
- [3] "Docker." [Online]. Available: <https://www.docker.com/>
- [4] "Rust Programming Language." [Online]. Available: <https://www.rust-lang.org/>
- [5] "Axum." [Online]. Available: <https://github.com/tokio-rs/axum>
- [6] "SQLx." [Online]. Available: <https://github.com/launchbadge/sqlx>
- [7] "Svelte." [Online]. Available: <https://svelte.dev/>
- [8] "Tailwind CSS." [Online]. Available: <https://tailwindcss.com/>
- [9] "shadcn/ui." [Online]. Available: <https://ui.shadcn.com/>

## APPENDIX

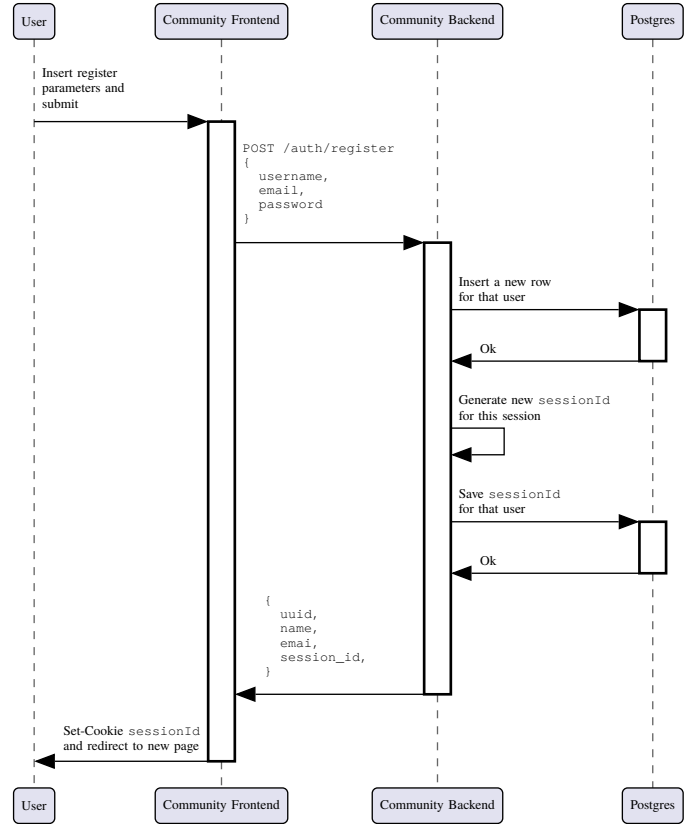


Fig. 1. Sequence Diagram of user registration with simple auth

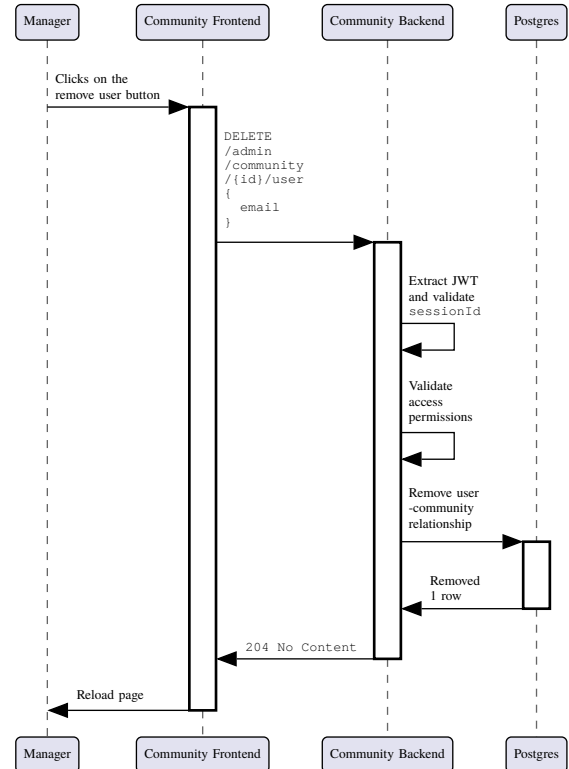


Fig. 2. Sequence Diagram for removing a user from community

## A. Community Database Schema

TABLE I  
DATABASE TABLE USER

Field	Type	Constraint	Description
id	UUID	Primary Key	The identifier of the user
name	VARCHAR	Not Null	The name of the user
email	VARCHAR	Unique	The email of the user
isAdmin	BOOLEAN	Not Null	Determines if the user is admin

TABLE II  
DATABASE TABLE KEY

Field	Type	Constraint	Description
provider	One of email, github or google	Primary Key	Authentication provider
id	VARCHAR	Primary Key	Identifier inside the provider namespace
userId	UUID	Foreign Key user.id	Owner of this auth credential
hashed Password	VARCHAR		Hash for password-based providers (empty otherwise)

TABLE III  
DATABASE TABLE SESSION

Field	Type	Constraint	Description
id	UUID	Primary Key	Session identifier stored in cookies
userId	UUID	Foreign Key user.id	User associated with this session
expiration	TIMESTAMPZ	Not Null	When the session becomes invalid

TABLE IV  
DATABASE TABLE COMMUNITY

Field	Type	Constraint	Description
id	UUID	Primary Key	Identifier of the community
name	VARCHAR	Unique	Name of the community
description	TEXT		Long-form description for members
image	VARCHAR		Optional URL of the community image

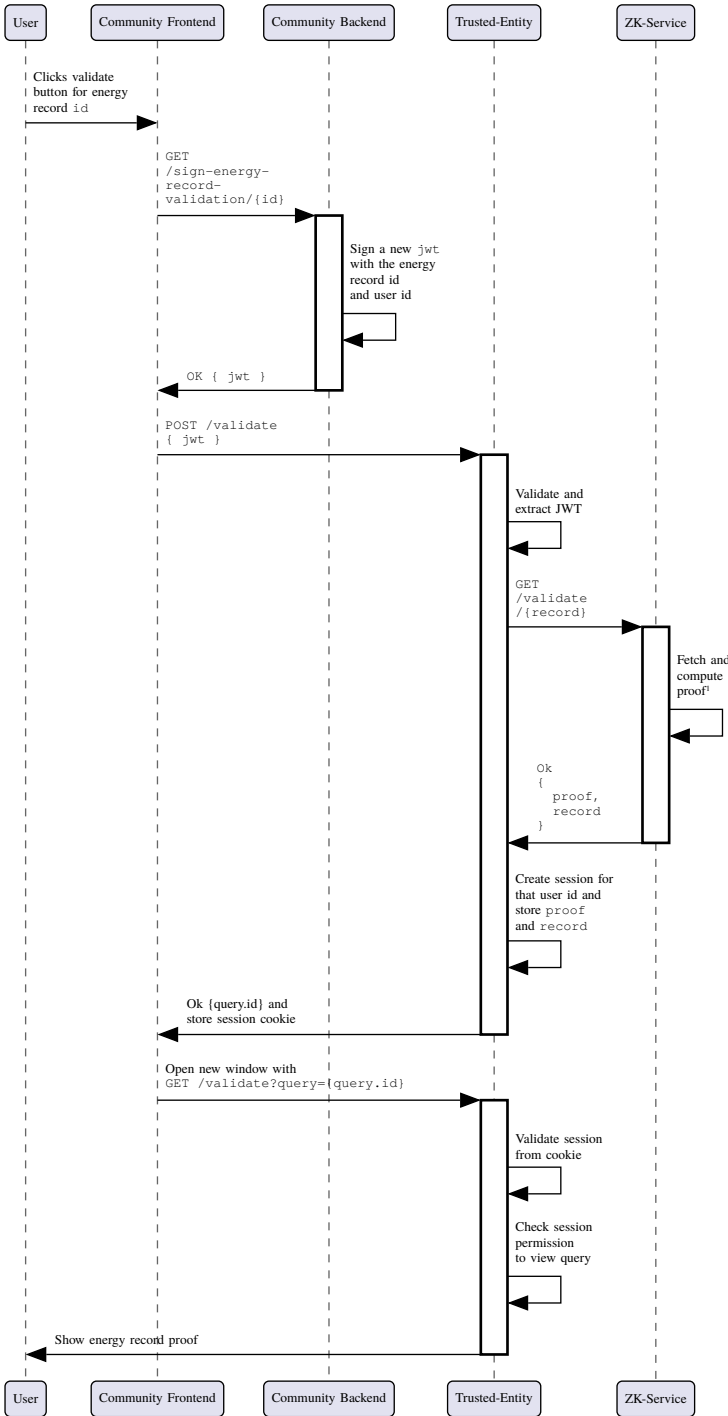


Fig. 3. Sequence Diagram for validate energy record

<sup>1</sup>In the current implementation this is done by querying the Community Backend directly as the ZK database was not ready.

TABLE V  
DATABASE TABLE COMMUNITY\_USER

Field	Type	Constraint	Description
userId	UUID	Primary Key and Foreign Key user.id	Member that belongs to the community
communityId	UUID	Primary Key and Foreign Key community.id	Community to which the user belongs

TABLE VI  
DATABASE TABLE COMMUNITY\_MANAGER

Field	Type	Constraint	Description
userId	UUID	Primary Key and Foreign Key user.id	Manager with elevated rights
communityId	UUID	Primary Key and Foreign Key community.id	Community which it refers to

TABLE VII  
DATABASE TABLE ENERGY\_RECORD

Field	Type	Constraint	Description
id	UUID	Primary Key	Identifier of the entry
userId	UUID	Foreign Key user.id	User associated with this record
communityId	UUID	Foreign Key community.id	Community context of the record
generated	NUMERIC		Energy generated (kWh)
consumed	NUMERIC		Energy consumed (kWh)
consumerPrice	NUMERIC		Price paid by the consumer
sellerPrice	NUMERIC		Price received by the seller
start	TIMESTAMP	Not Null	Start timestamp of the recorded interval (every 15 minutes)