Faculty Computer Science and Engineering Hochschule Esslingen

Bachelor Thesis in Software Engineering and Media Computing

Private Information Retrieval of Location Data

Alain Cascán Zalewska

Supervisor:
Prof. Dr. Dominik Schoop

Second Examiner:
Prof. Dr. rer. nat. Gabriele Gühring

UNIZAR Supervisor:
Francisco Javier Fabra Caro

Date: September 8, 2025

Acknowledgements

I would like to express my gratitude to **Prof. Dr. Dominik Schoop**, my thesis supervisor, for his guidance, availability, and constructive feedback throughout the project, and to **Prof. Dr. rer. nat. Gabriele Gühring** for the time devoted to review the document.

I would like to thank my parents for their support and encouragement, and my partner for her patience and help.

Finally, I would like to thank **Alain Villagrasa**, **Garikoitz Arellano**, and **Daniel Herce** for the four years we shared during the degree.

Abstract

Location-based services (LBS) give useful tips such as “restaurants near me”, but they also expose a user’s exact GPS position with other personal data to the provider. This thesis studies how to answer those queries while the server does not learn anything about either the location or the search filters. Two private information retrieval (PIR) solutions are being built that run on homomorphic encryption, using the CKKS scheme at the 128-bit security level. The first solution splits the work between two non-colluding cloud servers, one holding the data and the other the secret key. The second solution keeps everything on a single server that never sees any secret key. Both store 20 000 encrypted restaurants from OpenStreetMap.

Experiments demonstrate a clear trade-off. The two-server design returns exact results in a few seconds using roughly 1 GB of RAM, since one server can decrypt intermediate values for distance checks. The single-server design enhances privacy by keeping the key client-side; however, because all comparisons remain encrypted and require bootstrapping, runtime increases to tens of minutes and memory usage to tens of gigabytes. Both solutions maintain an average deviation below 5 meters from the original restaurant locations and satisfy all stated privacy requirements.

All functional requirements (GR1 multi-attribute queries; GR2 $<$ 15 m mean deviation of the resulting point from the original one) and privacy requirements (PR1 no parameter leakage; PR2 $\geq$ 128-bit strength; PR3 TLS transport) are met. However, scaling to many simultaneous clients exposes a bottleneck. Each extra user uploads new ciphertexts, so RAM and CPU rise almost linearly; in the single-server layout the footprint can exceed a machine’s capacity and delay responses beyond commercial limits. Without compression, sharding or hardware acceleration, mass deployment would demand prohibitively large infrastructure.

These results prove that CKKS already enables feasible privacy-preserving points-of-interest search for low-concurrency scenarios. Choosing between the two architectures means weighing lower latency against maximal confidentiality.

Contents

Chapter 1

Introduction

Data privacy and confidentiality have become critical as services move to the cloud as shown in Figure 1, since cloud servers can now analyze an application's code and infer user data from query patterns. This capability enables providers (or attackers who gain access to their infrastructure) to reconstruct sensitive information such as location history or personal preferences, undermining user anonymity and exposing services to compliance violations, reputation damage, and potential legal liability.

In this context, protocols for Private Information Retrieval (PIR) prevent a server from learning which data items a client requests and more generally protect server-side data and client-side queries by ensuring unlinkability (so that individual requests cannot be correlated), unobservability (so that an observer cannot determine whether a query occurred) and anonymity (so that no request can be linked to a client identity). Conversely, identity denotes the binding of a query to a specific client via credentials and when present nullifies unlinkability and anonymity. To maintain privacy in PIR schemes, queries must carry no identity information and any required authorization should take place over a separate channel that never conveys PIR traffic.

The first challenge lies in concealing query contents. Cryptographic techniques

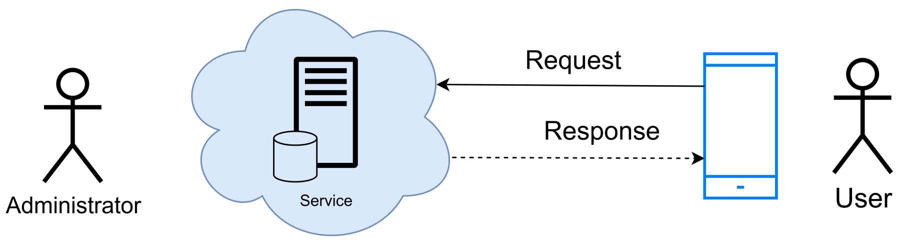


Figure 1: Typical architecture of a cloud service

attach additional data to each request and increase the size of the message. On databases measured in gigabytes, this overhead may remain manageable. As database sizes grow into terabytes or petabytes, the added data can saturate network bandwidth and render classic PIR schemes impractical. The second challenge arises in preventing request correlation and detection. Systems inject dummy queries to mask genuine requests and route traffic through mix nets to break observable patterns. Each additional mechanism increases bandwidth consumption and server processing demands and turns a simple lookup into a resource-intensive task.

When location-based data is involved, client privacy requires hiding not only the requested record but also the user's location. In the case of Points of Interest (POIs) such as restaurants or monuments, even minimal information leakage can expose travel patterns or frequently visited spots. Incorporating location privacy within a PIR-like framework adds constraints on communication cost and computation: queries must be generated so that servers cannot infer coordinates, and responses must be delivered without disclosing which region was queried. Achieving unlinkability, anonymity and unobservability in this setting may involve different techniques, each of which impacts latency and storage overhead. Addressing these limitations is essential in order to build a system that supports scalable, privacy-preserving retrieval, whether for POIs or other sensitive data in real-world deployments.

1.1 Task description

In this thesis, various Private Information Retrieval schemes are compared with the aim of preventing a server from distinguishing which data items a client requests. To this end, existing work on PIR, zero-knowledge databases and homomorphic encryption is consulted in search of potential solutions.

First, the problem will be introduced by explaining its context and detailing user needs and concerns in location-based services. These needs are then clearly identified and existing proposed solutions are gathered for review. Then the collected material is analyzed to reveal the strengths and weaknesses of current approaches, informing the search for areas of improvement, and sparking ideas for a more effective design.

Building on these insights, the requirements for the proposed solution are defined in terms of general criteria, privacy objectives and the attacker model, thus laying a solid foundation. Selected schemes are then designed and implemented according to these specifications. Once implementation is complete, testing with real data enables a comparison of the schemes in metrics such as speed, memory consumption and other relevant parameters, leading to the final conclusions.

1.2 Document structure

The thesis is structured as follows: Chapter 2 introduces the key concepts needed for the design and implementation of the existing and proposed solutions. Then, Chapter 3 provides a summary of the reviewed literature, describes the use case and the scenario, and outlines the general requirements, privacy requirements, and the attacker model on which everything will be built. Chapter 4 covers the conceptual design and implementation, including key decisions made during development. Chapter 5 presents the performance evaluation and relevant metrics, while Chapter 6 concludes with an overall conclusion of the project.

Chapter 2

Technical background

The chapter outlines the necessary technical background to understand the mechanisms applied within this work. It starts by introducing the homomorphic encryption protocol, a method of computation over ciphertext without compromising privacy. It presents various types of homomorphic schemes such as fully, partial, and somewhat homomorphic and highlights the CKKS scheme, since it allows approximate arithmetic and is optimized for practical applications of real or complex inputs.

To define the cryptographic infrastructure, this chapter also describes the problem of Ring Learning With Errors (RLWE), a lattice-based hardness assumption that offers security guarantees of the highest quality even against quantum attacks. RLWE offers efficiency along with security and thus remains a central part of contemporary homomorphic encryption schemes.

The later parts of the chapter deal with the concepts of geolocation. The chapter discusses how GPS coordinates are used to express positions on Earth and two popular distance models: the Haversine formula which is suitable for global distances and the Euclidean formula ideal for computation at much closer scale.

2.1 Homomorphic Encryption

Homomorphic encryption involves a cryptographic method that enables mathematical computations on encrypted information without decrypting it at any stage. The central goal is that the decrypted output should be precisely the same as the one that would have resulted if such computations had been executed on the plaintext information.

The initial idea was formulated in 1978 by Rivest, Adleman, and Dertouzos on the theme of carrying out computations on encrypted information [34]. It was then only a theoretical proposal since there was no encryption method at the time.

To better understand homomorphic encryption, the following examples are presented:

Example 1: Homomorphic Addition

1. **Initial encryption**: A user encrypts two numbers, for example, $a = 2$ and $b = 3$, obtaining ciphertexts $Enc(a)$ and $Enc(b)$.
2. **Operation on ciphertexts**: Without knowing the actual values, a third party performs a homomorphic addition:
$$Enc(a) \oplus Enc(b)$$
3. **Decrypting the result**: Using their private key, the user decrypts the ciphertext and obtains:
$$Dec(Enc(a) \oplus Enc(b)) = 2 + 3 = 5$$

Example 2: Homomorphic Multiplication

1. **Initial encryption**: A user encrypts two numbers, such as $x = 4$ and $y = 5$, resulting in ciphertexts $Enc(x)$ and $Enc(y)$.
2. **Operation on ciphertexts**: A third party performs a homomorphic multiplication:
$$Enc(x) \otimes Enc(y)$$
3. **Decrypting the result**: The user decrypts the ciphertext using their private key and obtains:
$$Dec(Enc(x) \otimes Enc(y)) = 4 \times 5 = 20$$

2.2 Homomorphic Cryptographic Families

Homomorphic encryption schemes are divided into various cryptographic families according to the following capabilities and limits:

1. Fully Homomorphic Encryption (FHE):

   Enables any number of addition and multiplication operations over ciphertexts and thus supports computation on encrypted data in general. Although it offers major theoretical and practical advantages, Fully Homomorphic Encryption is very expensive computationally because of the complexity of managing noise.

2. Partially Homomorphic Encryption (PHE):

   Enable unlimited operations for a single type of operation (either addition or multiplication, but not both). The Paillier encryption scheme [29] is a well-known example that allows unlimited additions but not multiplications.

3. Somewhat Homomorphic Encryption (SWHE):

   Allows a limited number of addition and multiplication operations on encrypted information, constrained by noise accumulation that eventually blocks further computations. It is generally used when the number of operations is known ahead of time and continues to be small. This scheme relies on FHE for its implementation.

2.3 Ring Learning With Errors (RLWE)

The Learning With Errors (LWE) problem underlies a fundamental assumption in modern cryptography. It relies on the premise that solving a certain class of noisy linear equations is a hard problem. More specifically, the adversary is given a set of equations of the type:

$$\mathbf{b}_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \mod q$$

where each $\mathbf{a}_i$ is a known vector, $\mathbf{s}$ is a secret vector, and e_i is a small error sampled from some noise distribution. Although this appears to be a linear equation system, the noise prevents us from easily solving it. The question is: Is it possible to recover the secret vector $\mathbf{s}$ from many such noisy samples?

The significance of LWE derives from its worst-case reduction of hard lattice problems such as the Shortest Vector Problem (SVP), which implies that the average solution of LWE is as challenging as solving the hardest instances of these lattice problems [33]. This makes LWE a strong basis for cryptography.

Building on LWE, the Ring Learning With Errors (RLWE) problem was formulated to make computation more efficient and decrease key sizes by exploiting the algebraic properties of polynomial rings [22]. In RLWE, the secret and inputs are not vectors, but polynomials in a ring $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$, often where $f(x)$ is a cyclotomic polynomial.

Instead of working with scalar inner products, RLWE samples take the form:

$$(a(x),\ b(x) = a(x) \cdot s(x) + e(x)) \in R_q \times R_q$$

Here, $a(x)$ is sampled uniformly at random, $s(x)$ is the secret polynomial, and $e(x)$ is a small error polynomial, usually with small coefficients. The security assumption is that, even with a large number of such samples, recovering $s(x)$ is computationally difficult. The decision version of the problem asks whether a given pair $(a(x), b(x))$ was generated in this way or was chosen uniformly at random.

RLWE maintains the strong theoretical security of LWE while allowing more efficient and compact cryptographic constructions thanks to its ring-based formulation. subsequently, RLWE is a great fit for building real-world cryptographic systems like lattice-based encryption schemes, digital signatures, and homomorphic encryption.

Another advantage of RLWE-based schemes is that they are compatible with *Single Instruction Multiple Data* (SIMD) operations. This technique allows multiple encrypted messages to be packed into a single ciphertext and processed at the same time, increasing its efficiency. This technique is especially powerful in homomorphic encryption systems [4], where SIMD-style batching allows operations on vectors of encrypted data with only a small overhead compared to regular scalar operations.

Example

Let's set the ring $R_q = \mathbb{Z}_{17}[x]/\langle x^2 + 1\rangle$, and suppose the secret is $s(x) = 1 + 2x$. Choose a random polynomial $a(x) = 3 + x$ and a small error $e(x) = 2$. Then the value of $b(x)$ is computed as:

$$b(x) = a(x) \cdot s(x) + e(x) = (3 + x)(1 + 2x) + 2$$

Expanding the product:

$$b(x) = 3 + 6x + x + 2x^2 + 2 = 3 + 7x + 2x^2 + 2$$

Since we are in the ring where $x^2 \equiv -1$, we substitute $2x^2$ with -2:

$$b(x) = 3 + 7x - 2 + 2 = 3 + 7x$$

So the final sample is $(a(x) = 3 + x,\ b(x) = 3 + 7x)$, which hides the secret $s(x)$ through the addition of a small error.

2.4 CKKS Scheme

CKKS [6] is a somewhat homomorphic encryption scheme designed to support approximate arithmetic on real or complex numbers. Unlike classic homomorphic encryption methods that work exclusively with integers or binary messages, CKKS encodes floating-point values by scaling them to integers and embedding them into polynomials. CKKS depends on the RLWE problem's hardness that the attacker cannot decrypt the information under plausible computational abilities without having the secret key. In practice, RLWE introduces carefully structured noise into the ciphertext, which protects the plaintext but also requires precise parameter choices to manage error accumulation.

The idea behind CKKS is to allow homomorphic addition and multiplication on encrypted data while accepting a small error of approximation. Such error naturally results from the fixed-point representation (determined by a *scaling factor*) as well as the lattice-based noise that contributes to security. Although CKKS tends to be classified as a somewhat homomorphic encryption scheme, multiple consecutive operations are supported as long as the parameters are set such that noise stays within reasonable bounds. In reality, when ciphertexts are multiplied together, additional steps (such as *relinearization* and *rescaling*) are applied to maintain coefficients and noise at manageable levels. By tailoring these parameters to the target application, CKKS can achieve a practical balance between performance, security, and numerical accuracy.

Below are the main parameters that need to be configured to use CKKS effectively, followed by a brief demonstration of homomorphic addition and multiplication.

Scaling Factor (Δ) This parameter controls how many fractional bits of precision are kept when converting real or complex values into polynomial coefficients. Before encryption, each value is multiplied by Δ and treated as an integer for the lattice-based computations. After decryption, the output is divided by Δ to recover the approximate plaintext.

- *High Δ:* Keeps more decimal places during computations, improving accuracy. However, bigger numbers can cause noise to grow faster, which might limit the number of valid multiplications before rescaling is needed.
- *Low Δ:* Reduces precision but slows down noise buildup, allowing deeper computations without exceeding noise limits.

Ciphertext Modulus (q) CKKS works with polynomials whose coefficients are taken modulo q. Typically, q may decrease through a process called *rescaling*, which triggers after certain operations, especially multiplications.

- *Larger q:* Allows more multiplications before the accumulated noise makes the ciphertext unusable. This is helpful for complex computations involving many steps.

- *Smaller q:* It lowers the storage and computational cost per operation but limits the total number of multiplications that can be performed in sequence.

Ciphertext Dimension (N) The dimension N (often a power of two) dictates the polynomial ring $R = \mathbb{Z}[X]/(X^N + 1)$ in which encryption takes place. It affects both security and performance:

- *Higher N:* Increases security as the underlying lattice problem (Ring-LWE) becomes harder to solve. However, it also increases memory usage and slows down computations.
- *Lower N:* Speeds up operations and makes ciphertexts smaller but reduces the security margin.

Batch Size CKKS can pack multiple real or complex numbers into a single ciphertext using the structure of the polynomial ring. The batch size is the maximum number of values that can be embedded at once.

- *Larger batch size:* Allows parallel processing of many encrypted values at once (e.g., applying operations element-wise on vectors). This is highly beneficial in data analytics and machine-learning use cases.
- *Smaller batch size:* Simplifies ciphertext management and can make implementations easier, but at the cost of processing fewer values at once.

Example of Homomorphic Operations in CKKS

Homomorphic Addition. Suppose $\mathbf{c}_x$ and $\mathbf{c}_y$ are CKKS encryptions of two real numbers x and y. The homomorphic addition of these two ciphertexts is:

$$\mathbf{c}_{\text{sum}} = \mathbf{c}_x \; + \; \mathbf{c}_y,$$

where the operation is performed on the polynomial coefficients. Decrypting $\mathbf{c}_{\text{sum}}$ with the secret key reveals an approximation to $x + y$. This addition typically requires no extra steps beyond the direct polynomial addition in the encrypted domain.

Homomorphic Multiplication. Let the same ciphertexts represent x and y. Their homomorphic product is:

$$\mathbf{c}_{\text{prod}} = \mathbf{c}_x \; \times \; \mathbf{c}_y.$$

Since multiplying two ciphertexts increases both the degree of the resulting polynomial and the internal scale, two subsequent operations are typically performed:

- *Relinearization:* Converts the product back into the standard two-part ciphertext format, making sure it stays compatible for further homomorphic multiplications or additions.
- *Rescaling:* Divides by a suitable factor (usually the same Δ) and adjusts q to keep the growth of coefficients under control. This keeps the scheme's internal noise at reasonable levels.

Decryption of $\mathbf{c}_{\text{prod}}$ yields a value close to $x \times y$, depending on the choice of Δ and how often rescaling is performed.

2.5 GPS Positioning

Global Positioning System (GPS) uses satellites and timing signals, but a main part of understanding how locations are determined is realizing how coordinates on Earth work.

Latitude and Longitude as Angular Coordinates [9] To picture a location on our planet, imagine slicing the Earth both horizontally and vertically as is represented in the Figure 2.

- **Latitude** (ϕ) shows how far north or south a point is from the Equator (considered $0°$). As you move toward the North Pole, latitude increases up to $+90°$; moving toward the South Pole, it drops to $-90°$.
- **Longitude** (λ) measures how far east or west a point is from the Prime Meridian (which passes through Greenwich, England, and is considered $0°$). Longitude ranges from $-180°$ in the extreme west to $+180°$ in the extreme east.

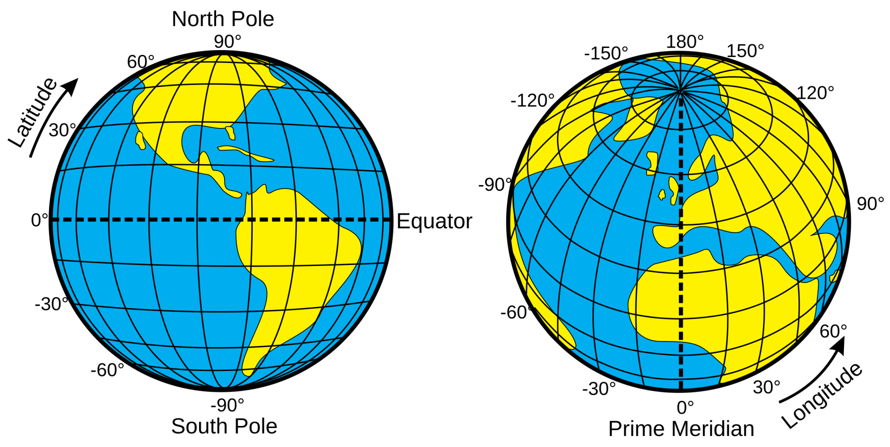


Figure 2: Latitude and longitude in the globe [10]

By using these angles together, any point on the planet can be identified. For most geolocation and mapping services, latitude and longitude are expressed in degrees, but many distance formulas require them to be in radians.

2.6 Euclidean Distance

When working with points of interest (POIs) located within relatively small geographic areas like a neighborhood, a university campus, or cities, using the Euclidean distance is usually a better choice than relying on more complex formulas like Haversine [23].

The Euclidean distance is simple, computationally efficient, and accurate enough for small-scale measurements where Earth's curvature can be ignored since its ignorable impact. At these smaller scales (typically less than a few kilometers), the errors introduced by treating Earth's surface as flat are imperceptible.

However, GPS coordinates (latitude and longitude) are usually given in degrees. To use the Euclidean distance correctly, these angular differences must first be converted into linear distances (in meters). The following approximations are commonly used for small geographic areas:

- Convert latitude difference to meters:

$$\Delta y \approx 111320 \cdot (\phi_2 - \phi_1)$$

- Convert longitude difference to meters (considering average latitude):

$$\Delta x \approx 111320 \cdot \cos\left(\frac{\phi_1 + \phi_2}{2}\right) \cdot (\lambda_2 - \lambda_1)$$

Where:

- ϕ_1, ϕ_2 are the latitudes of points 1 and 2 (in degrees).
- λ_1, λ_2 are the longitudes of points 1 and 2 (in degrees).
- 111320 is an approximate conversion factor representing meters per degree of latitude.

Once the differences Δx and Δy are obtained in meters, the Euclidean distance formula can be directly applied:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

Note that converting angular differences into meters is necessary only when precise real-world distances are required. If the goal is simply to filter POIs or check whether they fall within a certain range, directly using latitude and longitude degrees as approximate coordinates is enough.

This works because the Euclidean formula relies on comparing relative distances. Since latitude and longitude change smoothly and almost linearly over small regions, the ordering of distances and the inclusion within a fixed radius (in degrees) remains consistent.

Example: Suppose we have two POIs defined by GPS coordinates:

- **Library:** Latitude: 19.4333°, Longitude: -99.1400°
- **Cafeteria:** Latitude: 19.4350°, Longitude: -99.1415°

First, angular coordinates are converted to meters:

$$\Delta y = 111320 \cdot (19.4350 - 19.4333) \approx 189.24\, m$$
$$\Delta x = 111320 \cdot \cos\left(\frac{19.4333 + 19.4350}{2}\right) \cdot (-99.1415 + 99.1400) \approx -157.55\, m$$

Then, Euclidean distance is applied:

$$d = \sqrt{(-157.55)^2 + (189.24)^2} \approx 246.3\,m$$

Since the distance is relatively short, the Euclidean approximation provides an accuracy comparable to more advanced methods while being easier to implement and much faster to compute.

Therefore, for POI-related tasks where distances remain small, using the Euclidean formula combined with a basic angular-to-linear conversion offers the best trade-off between simplicity, speed, and accuracy. This makes it the preferred method in these situations.

Chapter 3

Problem Analysis and Requirements

As explained in Chapter 1, cloud servers that observe query patterns can deduce which records a user requests, thereby undermining unlinkability, unobservability and anonymity. The following section examines privacy-preserving techniques designed to prevent this inference and evaluates their respective strengths and weaknesses.

After the literature review, the chapter presents an example in which users query for nearby places without revealing their location. Using homomorphic encryption, the server evaluates the predicate (for example, 'within 500 meters') directly over the ciphertexts and returns only the matching results. This minimizes what the server learns about the user's coordinates, aside from potential leakage such as result size.

The chapter lists the system needs and the threat model. It specifies what the system should support (such as multi-attribute queries and strong encryption), the privacy conditions that must be guaranteed, and the type of attackers it aims to protect against.

3.1 Use Case

Location-Based Services (LBS) leverage a user's geographical position to provide personalized recommendations for nearby points of interest, such as restaurants, monuments or parks, often applying filters based on user preferences. However, sharing precise GPS coordinates with service providers raises significant privacy concerns. The storage and analysis of such sensitive data can reveal movement

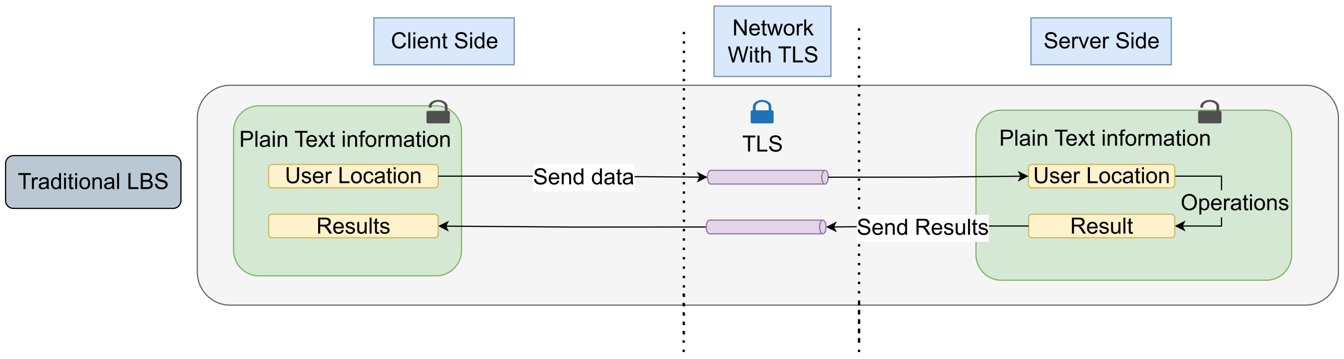


Figure 3: Traditional LBS

patterns, enable user profiling, facilitate cross-referencing with external databases for identity inference, and support targeted advertising. In the worst-case scenario, if the server is compromised, an attacker could reconstruct a user's history and use it for malicious purposes such as extortion.

Consider a user exploring a city who wants to find restaurants within a certain radius, below a specified price cap, offering vegetarian options and selected cuisines. In the traditional LBS model shown in Figure 3 the client sends its raw GPS coordinates and filter parameters to the server over a channel protected by TLS. The server decrypts the data, computes distances to candidate venues, applies the price and cuisine filters and returns the matching list. Although TLS protects the data in transit, the server still learns the user location. However, this is problematic as very sensitive location data is exposed to the service provider, failing to achieve anonymity, unlinkability and unobservability.

3.2 Literature summary

Three major privacy-preserving strategies have evolved to address different aspects of exposing sensitive data: Private Information Retrieval (PIR), location privacy methods for Location-Based Services (LBS), and encrypted databases relying on advanced cryptography such as homomorphic encryption or zero-knowledge proofs.

1. Private Information Retrieval (PIR):

   PIR is designed so that a user can retrieve an item from a database without revealing which item they are interested in. Early proposals relied on multiple independent servers that held copies of the database [7]. By distributing queries across those servers in a way that each server sees only a masked fragment of the request, the user obtains the desired data without a single server learning which record was requested, provided that the servers do not collude. This model can achieve strong privacy guarantees (sometimes information-theoretic), but it also requires maintaining multiple synchronized copies of the database, which is often not feasible in practice.

Single-server PIR solutions rely on cryptographic hardness assumptions [21]. The server holds the database in cleartext and to make a query, the client builds a vector that looks random but hides the target position. The server processes this vector by combining it with the data using basic arithmetic and returns a single result. From this, the client can extract the value of interest without revealing which entry was requested. The server sees only an opaque query and a final number, learning nothing about the access pattern. Over time, research on single-server PIR has reduced both communication overhead and response times using techniques such as vectorized arithmetic and modular transforms [17]. Nevertheless, computational overhead for the server remains significant, and basic PIR solutions usually handle only index-based lookups, making them less suited for more elaborate queries (for example, filtering by range or combining multiple attributes).

PIR has already been integrated into real deployments such as in Oblivious DNS (ODNS), which inserts an intermediate proxy into the DNS resolution path. The stub resolver encrypts the domain name under a fresh session key and then encrypts that key for the proxy. The proxy forwards only the encrypted domain to the authoritative resolver, which replies without learning the original label. Privacy holds as long as proxy and resolver do not collude; the end-to-end latency increases by one extra round trip and the bandwidth by roughly ten percent [35].

The main constraint that remains is server workload. Each query makes the server scan almost the whole database, so CPU time and disk traffic per lookup grow with the database size. Any record added or removed breaks the pre-processed view and requires a full re-computation.

2. Location Privacy for LBS:

   When using Location-Based Services, users often are required to provide coordinates or other location data to receive relevant results (e.g., points of interest). Privacy-preserving LBS methods aim to protect the user's identity or exact position. A straightforward approach is to broaden the reported location into an area that contains multiple potential users, known as spatial cloaking or k-anonymity [15]. This method reduces the ability of a service to locate the user's real position but can lower the accuracy of results (since the user effectively queries a larger region).

   Another strategy involves sending false location data, or "dummy" coordinates, with the real query to confuse the service provider [20]. This approach is simple to implement and can be effective if the dummy coordinates are realistic enough to make it difficult for the provider to distinguish the real request. However, generating multiple plausible queries can increase the overall data traffic. Across all of these methods, there is an inherent trade-off between achieving strong privacy guarantees and providing accurate real-time results in a geographically distributed system [3, 19]. Formally, a location query can be modelled as PIR by treating the map cell as an index, but the converse is not true: generic PIR protects arbitrary index sets without relying on spatial structure. Location-privacy schemes exploit that structure (widening the reported region or inserting dummy points) so the server does less work and bandwidth stays low, yet some coarse location information and cross-query links still leak. These methods are therefore a faster but weaker alternative: best described as *PIR-inspired* or an *approximate PIR*, because their security guarantees are strictly lower than those of cryptographic PIR.

3. Encrypted Databases (Homomorphic Encryption & Zero-Knowledge Proofs):

   Encrypted database systems aim to protect not only the user's query, but also the underlying records from an untrusted storage or processing environment. In a fully homomorphic encryption (FHE) framework [12], the server operates on ciphertexts, performing additions and multiplications that, when decrypted, yield the correct result as if computed on plaintext data. This feature theoretically allows for arbitrary queries on encrypted data while preventing the server from learning anything about the stored records. However, fully homomorphic operations typically require several orders of magnitude more computation than plaintext operations, which can pose substantial performance challenges for large or complex queries [36]. In our case, the data are public, so our objective is not to conceal the information but to guarantee the integrity and immutability of all operations, while also ensuring anonymity, unlinkability and unobservability.

   Partial or leveled homomorphic encryption may offer performance improvements by supporting only a limited set or depth of operations, suitable for specific query types such as basic arithmetic [31]. Even then, these systems often involve intricate key management and introduce overhead that may be prohibitive if the data or query volume is large.

   Zero-knowledge proofs guarantees that the prover leaks no information beyond the truth of the statement being proven, thus it protects the witness presented by the prover. The verifier, however, does not automatically gain privacy for its own inputs (additional protocol design is required such as secure two-party computation). However, a protocol can achieve data privacy (as in PIR) without supplying a zero-knowledge guarantee, nothing forces the server to prove correct execution. Therefore, zero-knowledge is sufficient but not necessary for many privacy goals, and privacy alone does not imply zero-knowledge.

For example: A city council collects votes encrypted under fully homomorphic encryption. A server homomorphically adds the ciphertexts and publishes two artifacts: (i) **S**, the total encrypted with the same public key, and (ii) a non-interactive zero-knowledge proof demonstrating that there exists a list of ciphertexts whose decryption yields only 0 or 1, whose clear-text sum equals n, and whose re-encryption under the public key equals **S**. Any observer, given the public key, **S**, and the proof, runs the public verifier, which checks a fixed set of algebraic equations and accepts only if all hold.

Each of these privacy-preserving techniques seeks to minimize the information a server or external observer can derive about user queries or stored data, but they address different priorities, which are briefly compared in Table 3.1.

Strategy	**Main mechanism**	**Cost / limitation**
Private Information Retrieval (PIR)	Multi-server: split masked request across replicas. Single-server: encrypt query, server applies algebraic operations.	Server scans almost the full database;
Location Privacy for LBS	Spatial cloaking (k-anonymity) or dummy coordinates that hide the real point.	Lower accuracy (cloaking) or higher traffic (dummies).
Encrypted Databases (FHE / ZKP)	Server computes on ciphertexts; zero-knowledge can add correctness proofs.	Orders-of-magnitude CPU overhead; complex key management; verifier privacy not automatic.

Table 3.1: Comparison of the three privacy-preserving techniques.

3.3 Homomorphic Encryption as a Privacy Enabler

An ideal LBS must deliver strong privacy guarantees without sacrificing functionality, and Homomorphic Encryption (HE) fulfills this by allowing servers to perform all computations on encrypted data. Figure 4 illustrates the HE-enhanced workflow: the client first encrypts its GPS coordinates and sends only ciphertext over a TLS channel; the server then performs distance calculations, category filtering, and ranking directly on these encrypted values, and returns the resulting ciphertext; finally, the client decrypts the response locally to obtain precise, personalized recommendations, while its raw location remains concealed at all times.

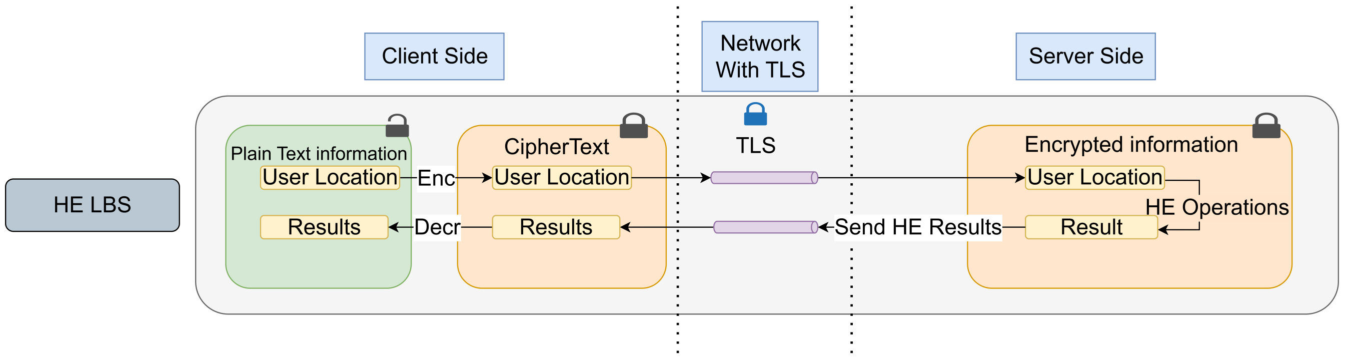


Figure 4: HE LBS

This homomorphic approach extends to a variety of real-world scenarios. Civil protection alerts can be delivered based on encrypted location data, notifying users of wildfires within a 10 km radius or nearby road accidents within 500 m, all without disclosing their whereabouts. Drivers seeking fuel or electric-vehicle charging points submit encrypted coordinates and receive filtered results by price, distance or charging power, preserving location confidentiality. Even in entertainment contexts such as augmented reality games, the server can provide position-based content and events by operating on encrypted locations, ensuring players' true positions remain hidden while delivering a rich interactive experience.

3.4 Requirements

Considering the use of the use case presented in Section 3.1, the general and specific privacy requirements as well as the attacker model will be defined in the following sections.

3.4.1 General requirements

General requirements refer to requirements related to general functionalities that the architecture must support:

GR1: The user will be able to perform multi-attribute queries by configuring continuous values (search radius in meters and numeric price) and discrete options (preconfigured cuisine types and vegetarian requirement yes/no).

GR2: The solution should not have a mean deviation between the original and the resulting points greater than 15 meters.

3.4.2 Privacy requirements

Privacy requirements refer to the conditions under which an individual exercises control over their personal data, including its collection, use, disclosure, and storage. These requirements arise from ethical principles, social norms, and other relevant sources.

PR1: The user query parameters must not be disclosed or inferred by any cloud server.

PR2: The encryption algorithm used must have a minimum security level of 128 bits.

PR3: All messages exchanged between the actors are transmitted over a secure channel (e.g. using TLS over TCP/IP).

Preventing any cloud server from accessing or inferring query parameters (PR1) eliminates explicit identifiers at the application layer, so messages do not carry tokens or user-specific values that could link them. Requiring 128-bit encryption (PR2) prevents pattern analysis or brute-force decryption. Transmitting messages over a secure channel (PR3), such as TLS over TCP/IP, conceals headers, parameters, and transport metadata from eavesdroppers. Together, PR1–PR3 remove both overt and covert linking vectors: with no cleartext parameters, no decipherable payload, and no exposed metadata, separate sessions become cryptographically isolated (unlinkability), and no distinguishing attributes remain to tie any message to a user (anonymity). This layered defense also ensures that, without keys or identifiers, adversaries cannot re-identify or correlate data, thus fully protecting user identity.

3.4.3 Threat Model

The following threat model specifies the trusted parties, the semi-honest (honest-but-curious) assumption, the adversary's capabilities, and provides a concise framework for the next phases.

- **Trusted parties**
  - All the actors will follow the protocol exactly and do not try to learn any extra data.
- **Semi-honest (honest-but-curious) assumption**

  - Cloud actors will log and analyse everything they see in order to learn secret information.
  - No collusion: we assume that all the servers held in the cloud are not colluding (e.g. held by different cloud providers).

- **Adversary capabilities**
  - Can *eavesdrop* on all traffic between every actor.
  - Can *compromise* at most **one** cloud actor.
  - Cannot break homomorphic encryption with security greater than or equal to 128 bits.
  - Cannot force the cloud servers to collaborate.

Threat-model assumptions, together with the general (GR1, GR2) and privacy (PR1, PR2, PR3) requirements, constitute the design framework for the next chapters, which develops concrete solutions tailored to these constraints.

Chapter 4

Solution Design and Implementation

A clear record is required to link the privacy goal (keeping a user's coordinates hidden) to code that enforces it. The text lays out each design choice and coding trade-off so every possible leak is visible and performance numbers rest on repeatable steps.

First, trust placement is clarified by comparing layouts with one cloud server and with two cooperating servers, each blocking a different kind of inference. Next, the decision to use the CKKS scheme is justified because its approximate arithmetic works on fixed-point coordinates without bit-level circuits. Finally, the engineering measures are shown: C++ for direct execution, OpenFHE for current homomorphic tools, vector packing for SIMD speed, and a controlled transport layer that removes network noise from timing tests.

4.1 Design

For the use case described in Section 3.1, the fist solution is going to be presented, where homomorphic encryption will take place to protect sensitive data, such as the user's location ad other parameters from cloud-based servers [16], while still allowing the system to retrieve nearby POIs that meet the user's constraints.

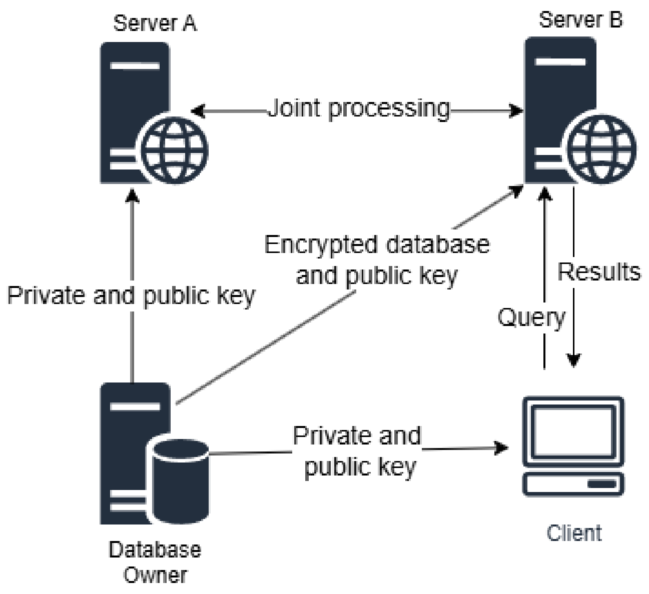


Figure 5: Overview of architecture 1

4.1.1 SOLUTION 1: Two cloud servers operating jointly

This solution will use two cloud-based servers and a key provider, where the Database Owner will be acting as a third-trusted party (TTP), which is in charge of holding the plain text database and generating the key pair, while the cloud servers are going to be cooperating in order to generate a result but with different roles that will be explained in the phases below. The diagram in Figure 5 shows the actors and illustrates the communication flows between all of them. Note that the figure shows a single client, but there could be several.

Phase 1: System initialization

As shown in Figure 6, the **Database Owner** first generates a homomorphic key pair, encrypts all data, and then pushes two different messages.

$$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}().$$

Let the clear database be arranged in attribute columns

$$\mathbf{v}_j = (a_{1,j}, \dots, a_{N,j}), \qquad j = 1, \dots, m,$$

which are encrypted as

$$c_j = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{v}_j), \qquad \mathcal{C} = (c_1, \dots, c_m).$$

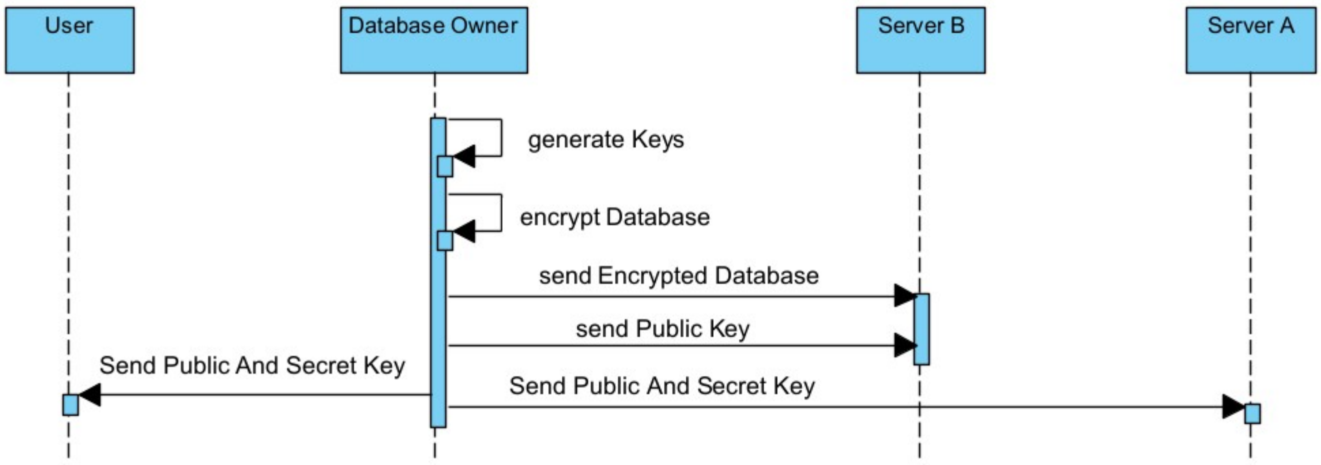


Figure 6: Key setup and database encryption with two servers

The outgoing messages are therefore:

$$\begin{aligned} M_B &= (\mathsf{pk}, \mathcal{C}) &\longrightarrow\ & \textbf{Server B} \\ M_A &= (\mathsf{pk}, \mathsf{sk}) &\longrightarrow\ & \textbf{Server A} \\ M_U &= (\mathsf{pk}, \mathsf{sk}) &\longrightarrow\ & \textbf{User} \end{aligned}$$

At this point, each actor holds only the material required for its role: **server B** can evaluate ciphertexts but never decrypt them; **server A** can decrypt and re-encrypt but never sees the raw database; the **user** can encrypt its queries and recover the answer.

Phase 2: User query

Figure 7 describes how **the user** first pack the location (ℓ), cuisine (γ) and vegetarian preference ($v \in \{0, 1\}$), maximum price allowed ($p_{\max}$), and maximum squared distance ($d^2_{\max}$) into a single plaintext vector

$$\mathbf{q} = (\ell, \gamma, v, p_{\max}, d^2_{\max}),$$

which is then encrypted once with the public key from Phase 1,

$$\tilde{q} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{q}).$$

Then the ciphertext is transmitted to **Server B**:

$$M_{U \to B} = \tilde{q}.$$

Afterwards, **Server B** and **Server A** will make some operations together creating a response, that will be sent back to the client

$$M_{B \to U} = \tilde{r}.$$

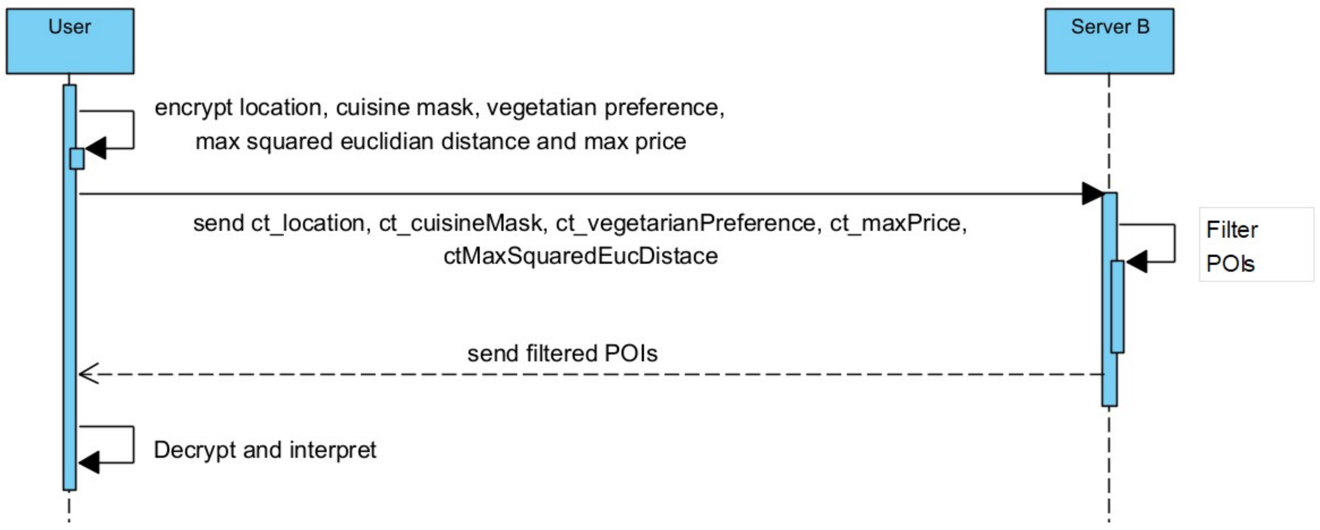

User
Server B
encrypt location, cuisine mask, vegetatian preference,
max squared euclidian distance and max price
send ct_location, ct_cuisineMask, ct_vegetarianPreference, ct_maxPrice,
ctMaxSquaredEucDistace
Filter POIs
send filtered POIs
Decrypt and interpret

Figure 7: Encrypted query submission and local filtering

The client then applies the secret key obtained during initialization to recover the plaintext result:

$$\rho = \mathsf{Dec}_{\mathsf{sk}}(\tilde{r}).$$

Phase 3: **Server B and Server A processing.**
Figure 8 depicts the interactive protocol that filters the encrypted database without disclosing raw data. Let N be the number of stored restaurants and define

$$\mathcal{C} = \big(c_i^{(\text{loc})}, c_i^{(\text{price})}, c_i^{(\text{veg})}, c_i^{(\text{cuisine})}\big)_{i=1}^{N}, \qquad \tilde{q} = (\tilde{\ell}, \tilde{\gamma}, \tilde{v}, \tilde{p}_{\max}, \tilde{d}^2_{\max}),$$

where $c_i^{(\text{loc})} = (\tilde{\phi}_i, \tilde{\lambda}_i)$ stores encrypted latitude (ϕ) and longitude (λ). The two servers perform:

(a) Squared–distance evaluation (**Server B**)

$$\tilde{d}_i = \mathsf{Eval}_{\mathsf{pk}}\big(\|c_i^{(\text{loc})} - \tilde{\ell}\|\big) \longrightarrow \text{Server A}.$$

(b) Distance squaring (**Server A**)

$$\tilde{d}_i^2 = \mathsf{Enc}_{\mathsf{pk}}\big(\mathsf{Dec}_{\mathsf{sk}}(\tilde{d}_i)^2\big) \longrightarrow \text{Server B}.$$

(c) Score computation (**Server B**)

$$\begin{aligned}\tilde{\Delta}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{d}^2_{\max} - \tilde{d}^2_i\big),\\ \tilde{v}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{v} \overset{?}{=} c_i^{(\text{veg})}\big),\\ \tilde{\pi}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{p}_{\max} - c_i^{(\text{price})}\big),\\ \tilde{c}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\mathsf{match}_{\tilde{\gamma}}\big(c_i^{(\text{cuisine})}\big)\big).\end{aligned}$$

Where $\mathsf{match}_{\tilde{\gamma}}(c_i^{(\text{cuisine})})$ returns the number of matching elements between the cuisine code c_i and the user cuisine preference $\tilde{\gamma}$.

(d) Coordinate blinding (**Server B**)
Choose random offsets $r_i^{\phi}, r_i^{\lambda} \leftarrow \mathbb{Z}_q$ where $\mathbb{Z}_q = \{10^8, \ldots, 10^9\}$ and define

$$(\tilde{\phi}_i^{\star}, \tilde{\lambda}_i^{\star}) = (\tilde{\phi}_i + r_i^{\phi},\ \tilde{\lambda}_i + r_i^{\lambda}).$$

Package

$$\tilde{s}_i^{\star} = \big(\tilde{\phi}_i^{\star}, \tilde{\lambda}_i^{\star}, \tilde{\Delta}_i,\ \tilde{\pi}_i,\ \tilde{c}_i,\ \tilde{v}_i\big) \longrightarrow \text{Server A}.$$

(e) Filtering (**Server A**)
After decryption, discard every i for which

$$\Delta_i < 0\ \vee\ \pi_i < 0\ \vee\ v_i = 0\ \vee\ c_i = 0.$$

Re-encrypt the surviving $(\tilde{\phi}_i^{\star}, \tilde{\lambda}_i^{\star})$ and return them.

(f) Deblinding (**Server B**)
Remove the random offsets:

$$(\tilde{\phi}_i, \tilde{\lambda}_i) = (\tilde{\phi}_i^{\star} - r_i^{\phi},\ \tilde{\lambda}_i^{\star} - r_i^{\lambda}),$$

then forward the cleaned ciphertexts to the user.

The primary weakness of this solution design is that the system's private key is stored on a cloud server over which we have no deep control (due to the semi-honest assumption). Consequently, if the threat model cannot be strictly enforced, overall security is significantly reduced, especially under the assumption that cloud servers might collude.

4.1.2 SOLUTION 2: One cloud server operating alone

After Solution 1 analysis, a second design is presented, shown in Figure 9. This solution removes **Server A** and the private key remains exclusively with the key provider and client, so no cloud server can decrypt data. The protocol preserves the homomorphic encryption workflow and the same filtering logic, while all remote computation is carried out by a single cloud server.

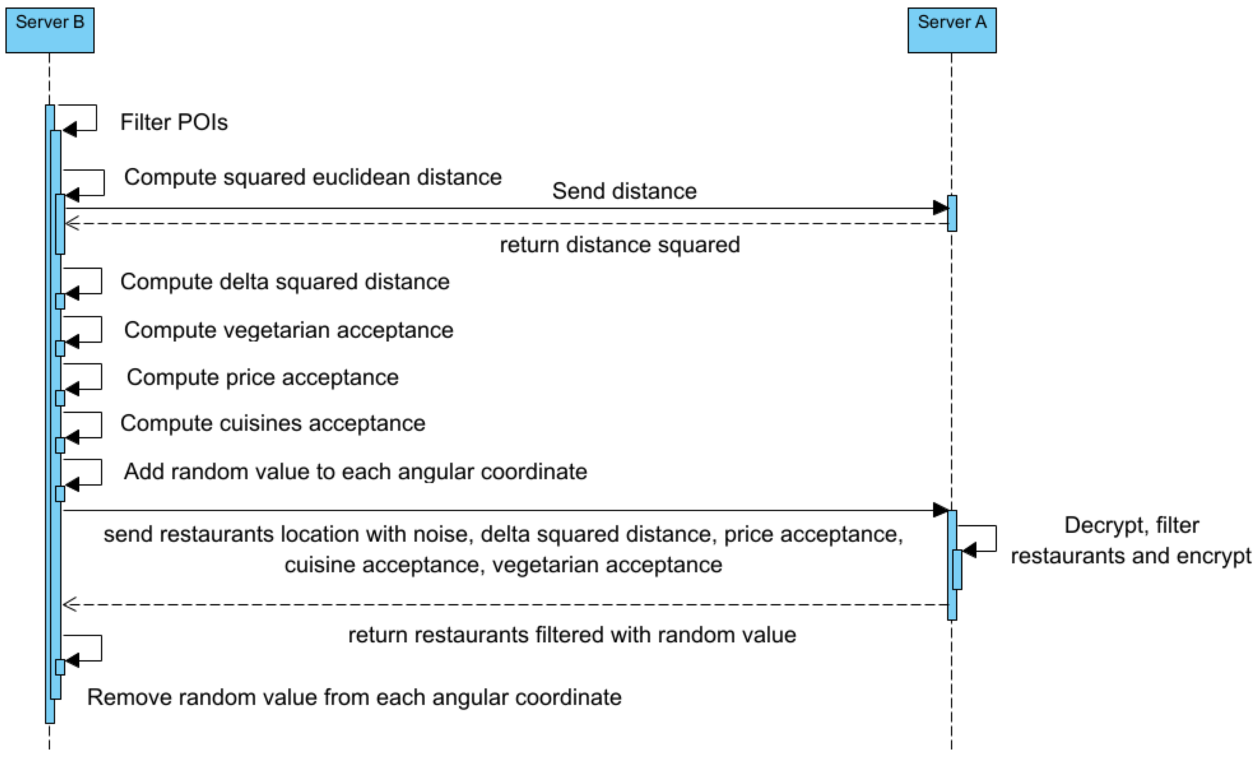

Server B
Server A
Filter POIs
Compute squared euclidean distance
Send distance
return distance squared
Compute delta squared distance
Compute vegetarian acceptance
Compute price acceptance
Compute cuisines acceptance
Add random value to each angular coordinate
send restaurants location with noise, delta squared distance, price acceptance, cuisine acceptance, vegetarian acceptance
Decrypt, filter restaurants and encrypt
return restaurants filtered with random value
Remove random value from each angular coordinate

Figure 8: Cooperative processing

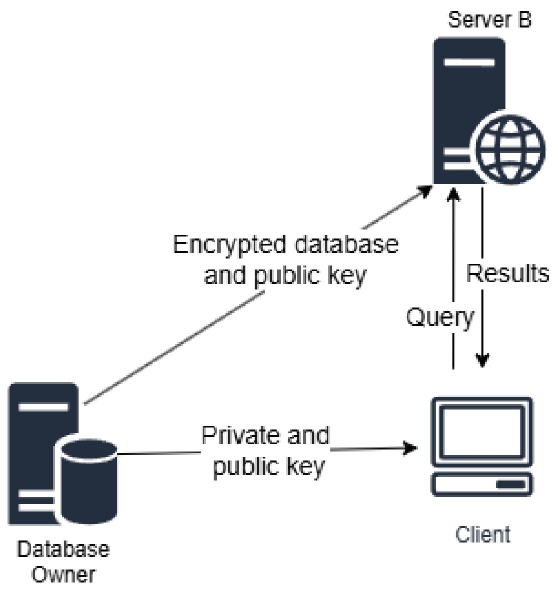


Figure 9: Overview of architecture 2

Phase 1: System initialization

This phase is identical to that of Solution 1 except that no data is ever sent to **Server A** as shown in Figure 10.

$$\cancel{M_A = (\mathsf{pk}, \mathsf{sk}) \longrightarrow \textbf{Server A}}$$

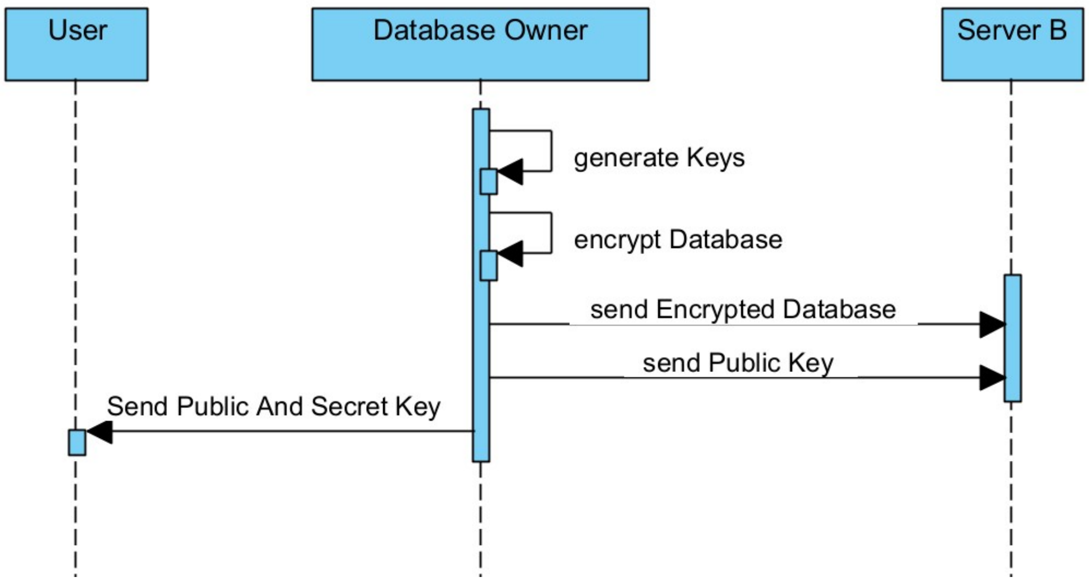


Figure 10: Key setup and database encryption with one server

Phase 2: User query

The process the user follows to query Server B remains the same as in Solution 1: using the public key obtained in the initial phase, the **user** encrypts its location, its cuisine and vegetarian preference, maximum allowed price and the squared maximum distance. The **client** then sends these encrypted values to **Server B** and since **Server B** already stores its database in encrypted form, it can perform operations between the **user** encrypted data and its stored data without revealing any information.

Phase 3: Server B processing

Now the process of filtering the restaurant points runs entirely on **Server B** as there is no other cloud server. The sequence of operations is shown in Figure 11. For every encrypted record

$$\mathcal{C} = \left(c_i^{(\mathrm{loc})}, c_i^{(\mathrm{price})}, c_i^{(\mathrm{veg})}, c_i^{(\mathrm{cuisine})}\right)_{i=1}^{N}, \qquad \tilde{q} = (\tilde{\ell}, \tilde{\gamma}, \tilde{v}, \tilde{p}_{\max}, \tilde{d}^2_{\max}),$$

with $c_i^{(\mathrm{loc})} = (\tilde{\phi}_i, \tilde{\lambda}_i)$, the server performs:

Feature	Solution 1: Single cloud servers	Solution 2: Double Cloud Server
Security Level	High	Very High
Flexibility	High	Moderate-low

Table 4.1: Comparison of the Two Solutions

(a) *Squared distance*

$$\tilde{d}_i^2 = \mathsf{Eval}_{\mathsf{pk}}\big(\|\tilde{\ell} - c_i^{(\mathrm{loc})}\|^2\big).$$

(b) *Raw acceptance scores*

$$\begin{aligned}
\tilde{\Delta}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{d}^2_{\max} - \tilde{d}_i^2\big),\\
\tilde{v}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{v} \overset{?}{=} c_i^{(\mathrm{veg})}\big),\\
\tilde{\pi}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\tilde{p}_{\max} - c_i^{(\mathrm{price})}\big),\\
\tilde{c}_i &= \mathsf{Eval}_{\mathsf{pk}}\big(\mathsf{match}_{\tilde{\gamma}}\big(c_i^{(\mathrm{cuisine})}\big)\big).
\end{aligned}$$

(c) *Filtering predicate*

$$\tilde{f}_i = (\tilde{\Delta}_i \geq 0) \;\wedge\; (\tilde{\pi}_i \geq 0) \;\wedge\; \tilde{v}_i \;\wedge\; (\tilde{c}_i > 0).$$

(d) *Selection* The encrypted subset

$$\big\{\, c_i^{(\mathrm{loc})} \;:\; \tilde{f}_i = 1 \big\}$$

which contains the points that meet the user's constraints, to the user for decryption.

All steps are evaluated homomorphically, no plaintext data are revealed to the cloud server.

Key Differences Between Solutions

Table 4.1 presents a brief comparison of the two previously proposed solutions. In terms of security level, both approaches are considered fully secure. However, the second solution is considered more secure, as it does not involve publishing the system's private key on a cloud-based server.

Finally, the first solution offers greater flexibility, as adding parameters for comparisons does not introduce major complexity or significantly increase computation time. In contrast, including additional parameters in the second solution would considerably increase computational costs.

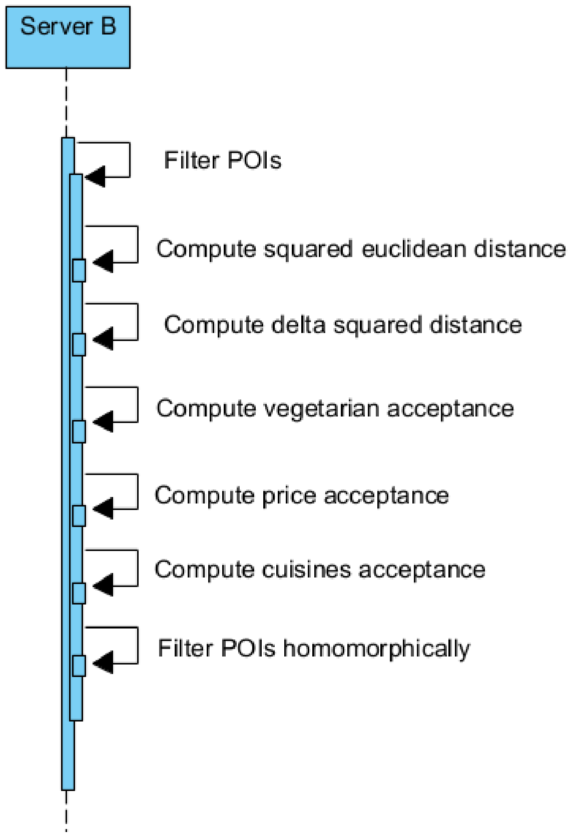

Server B
Filter POIs
Compute squared euclidean distance
Compute delta squared distance
Compute vegetarian acceptance
Compute price acceptance
Compute cuisines acceptance
Filter POIs homomorphically

Figure 11: Server B processing

4.2 Scheme Selection

Recent developments in fully homomorphic encryption (FHE) have increased the range of supported algebraic structures and performance metrics [14, 32]. Among these are the BGV and BFV schemes from 2012 which are based on the RLWE problem. They are well suited for implementation on low to medium depth circuits and enjoy strong library support like HElib, PALISADE, and SEAL [14]. However, their rapid noise growth along with exorbitant bootstrapping costs of several hundred seconds make deeper computations very inefficient.

FHEW (2015) and TFHE (2017/2020) shifted focus to the binary domain, applying LWE and torus-LWE, respectively. These schemes achieved remarkable increases in bootstrapping speed with TFHE attaining sub-second times (estimated 13 ms using AVX-512) [32]. However, these performance gains are limited to bitwise operations where every logic gate requires bootstrapping. Additionally, the absence of native SIMD packing greatly diminishes scalability for arithmetic operations.

Speaking of the CKKS scheme (Cheon-Kim-Kim-Song, 2017), the authors introduced an approximate representation of real and complex numbers over RLWE, making the scheme particularly useful for machine-learning and signal-processing tasks [18]. CKKS supports efficient vectorized operations on packed fixed-point values using SIMD, allowing streamlined addition, multiplication and rotation without the overhead of binary circuit conversion. Every multiplication undergoes modulus and scaling factor adjustments at a defined order: this helps control noise growth and precision loss simultaneously in a balanced manner considering accuracy and computational efficiency. In FHEBench benchmarking studies, CKKS is noted to deliver strong performance and high throughput for fixed-point arithmetic tasks [18]. Unlike BGV and BFV, CKKS does not require the scaling of real numbers into very large integers.

These strengths make CKKS the best candidate for working with real-world data such as coordinates of latitude and longitude. For this workload, its native support for real-valued numbers, vectorization and precision control is highly synergistic.

4.3 Implementation

Implementation decisions made both generally for the two selected architectures and specifically for each one will be presented. These decisions will be presented along with the reasoning behind each choice, explaining why they were made in this particular way.

4.3.1 General Implementation Decisions

Over the next subsections shared implementation choices relevant to both solutions are going to be presented. It covers the selected programming language and library, the method used for encoding the restaurant data, the communication approach between system components and how user-defined parameters will be used to filter points of interest.

4.3.1.1 Programming language and library

C++ was selected for this implementation as it compiles directly to machine code, removing Python's interpretation overhead and improving execution speed by avoiding bytecode handling and reducing function call latency. Additionally, C++ offers compiling checks for type mismatches and overflows, unlike Python where those issues typically appear during runtime, leading to delayed debugging and a higher error rate.

For the cryptographic library, OpenFHE [26, 30] was chosen as it provides multiple schemes (including BFV, BGV, and CKKS), offers better performance than other libraries such as Microsoft SEAL [11, 24], and natively supports bootstrapping. The latest version is v1.2.4, released on 21 March 2025. Supported by DARPA under a 2-clause BSD license, OpenFHE is built by contributors involved in the development of PALISADE, HElib, HEAAN, and FHEW, reinforcing its reliability in the field.

4.3.1.2 Data Codification

For this thesis, the work will focus on restaurant-related data: latitude, longitude, cuisine type(s) offered by each restaurant (users may select one or more from ten predefined options), average meal price, and a binary indicator denoting whether vegetarian options are available.

Each of these data elements will be stored in a separate vector of size N, where each component corresponds to a different restaurant, which will then be encrypted and serialized to create a database, as shown in Figure 12.

Following this architecture offers various advantages, such as modularity and scalability, which is beneficial if new restaurant filtering features are to be added in the future, as well as the ability to leverage the SIMD property of the CKKS algorithm to process the maximum number of restaurants per operation, and the possibility for users to filter by multiple restaurants at once, considering that a single restaurant may offer several types of cuisine.

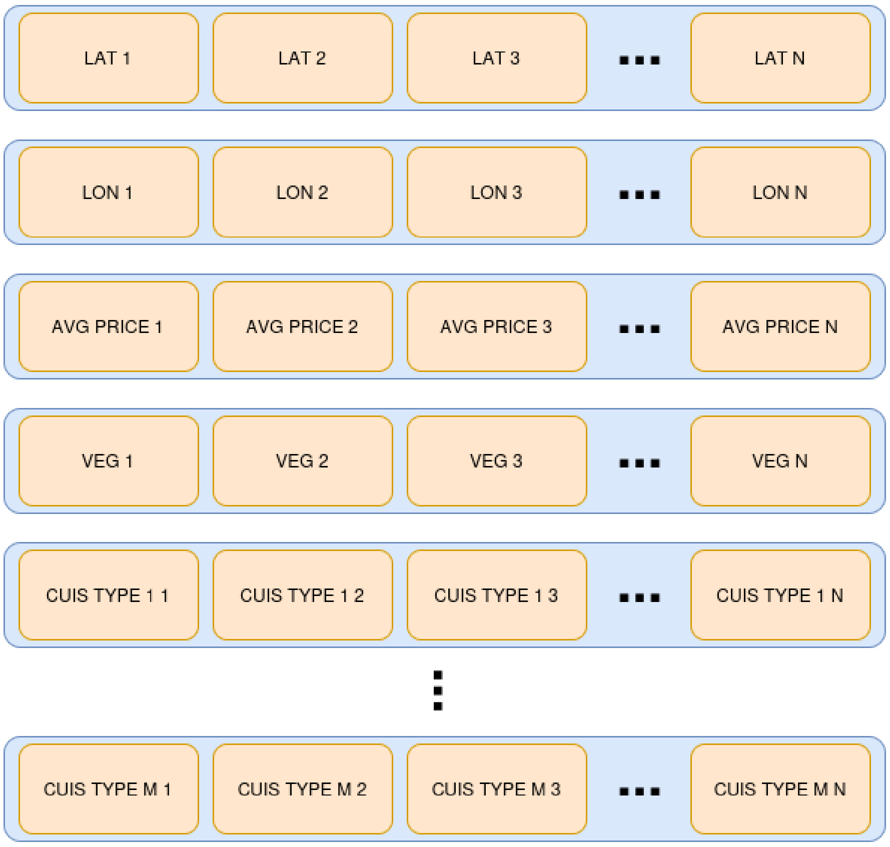


Figure 12: Data Codification

4.3.1.3 Data exchange between actors

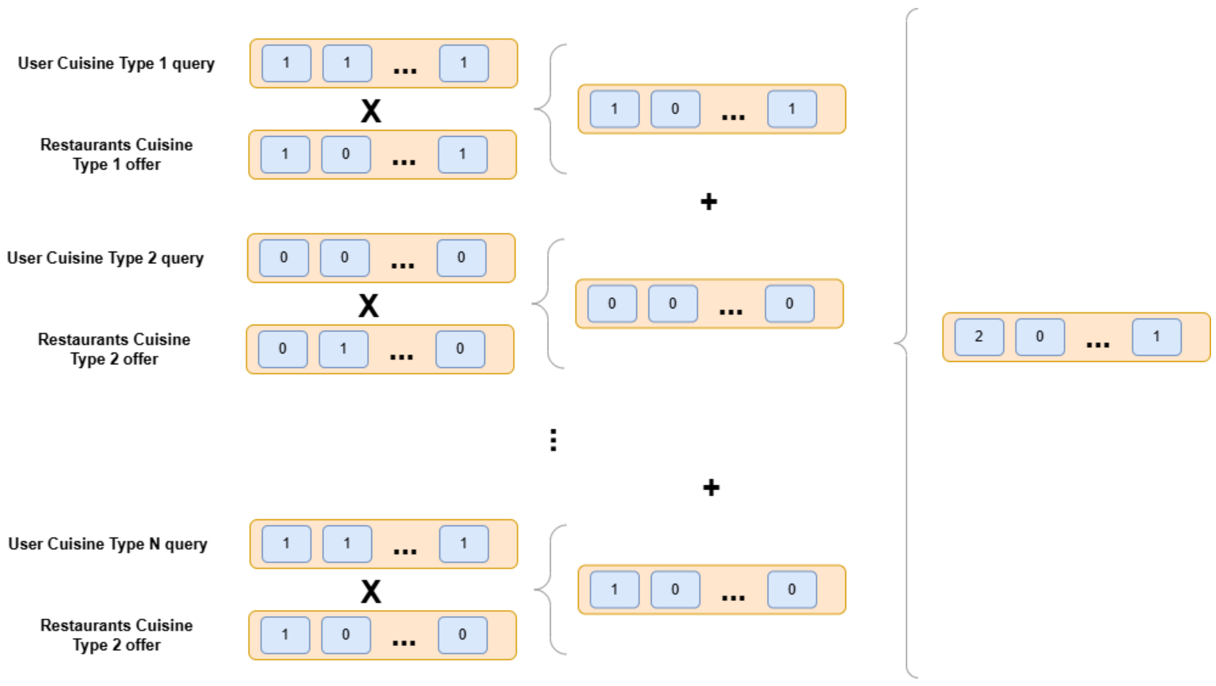
To manage the data exchange between the different actors, the addition of the TCP/IP model has been omitted, as its integration would add unnecessary complexity without providing relevant information, potentially introducing noise in the performance evaluation phase due to occasional network instabilities or other external factors. Therefore, to avoid timing variability introduced by TCP/IP and maintain reliability, the sender serializes data and the receiver deserializes it for each message exchange. This allows the evaluation to account for serialization and deserialisation times of the transmitted ciphertexts, while avoiding the inherent unpredictability of network conditions.

4.3.1.4 User Parameter-Based Vector Construction

To determine whether a restaurant meets the user-provided constraints, **Server B** generates a vector that serves as a filtering mechanism. Each solution uses these vectors differently: in the first approach, **Server A** decrypts and uses these vectors to perform the filtering while in the second architecture, the vectors are applied to multiply the POI, “disabling” it for the user by normalizing and carrying out a homomorphic comparison.

Vector creation depends on the parameter being evaluated. Below is the general method for vector generation (if there are specific architectural variations, they will be explained in the following sections):

- Distance Affinity Vector: To construct this vector, **Server B** computes the squared Euclidean distance homomorphically, allowing the determination of the distance of the user to each POI. After this computation, the **server B** subtracts the user-defined maximum acceptable distance, which the user has pre-converted to degrees before submitting it as a parameter. If any component of the resulting vector is negative, it indicates that the POI does not meet the distance requirement set by the user, while a positive value signifies that the POI satisfies the distance condition set by the user.
- Restaurant Affinity vector: This vector indicates whether a restaurant offers at least one type of cuisine that satisfies the user’s preferences. The user submits a query containing a two-dimensional vector, where each vector (corresponding to a specific cuisine) is filled with ones if the user is interested, or zeros otherwise. **Server B** then multiplies the user provided cuisine vector with the restaurant’s corresponding cuisine vector. A result of one indicates that there is a match between the user’s interest and the restaurant’s offering. On the other hand, a zero suggests that the restaurant does not offer that



User Cuisine Type 1 query
1 1 ... 1
X
Restaurants Cuisine Type 1 offer
1 0 ... 1
1 0 ... 1
+
User Cuisine Type 2 query
0 0 ... 0
X
Restaurants Cuisine Type 2 offer
0 1 ... 0
0 0 ... 0
2 0 ... 1
+
User Cuisine Type N query
1 1 ... 1
X
Restaurants Cuisine Type 2 offer
1 0 ... 0
1 0 ... 0

Figure 13: Cuisines Acceptance Process

cuisine or the user is not interested in it. This process is repeated for all available cuisine types, summing the results. A value greater than zero indicates the restaurant offers at least one cuisine of interest to the user, while a sum of zero means no match. This procedure is illustrated in the Figure 13

- Average price vector: This vector is calculated by subtracting the maximum average price accepted by a user from a restaurant's average price. Once this vector is obtained (if the value is positive then the restaurant meets the user's restriction; otherwise, it does not meet the condition).

- Vegetarian Acceptance vector: This vector requires a different calculation because a homomorphic XOR operation must be applied since the users can indicate that they want a vegetarian restaurant, that they do not want a vegetarian restaurant, or that they do not care if they are vegetarian or not. To perform this operation three parameters are required,p and m which are binary values (0 or 1) supplied by the user, and f which indicates whether the restaurant offers vegetarian food (Again encoded as a binary value 0 or 1). Then, an XOR is first applied using the formula

$$\text{XOR} = p + f - 2pf,$$

where $p = 1$ if the user wants a vegetarian restaurant (and $p = 0$ otherwise), and $f = 1$ if the restaurant offers vegetarian food (and $f = 0$ otherwise).

Once the XOR has been calculated, a NOT operation is applied:

$$e = 1 - \text{XOR},$$

obtaining $e = 1$ if $p = f$, and $e = 0$ otherwise. Finally, the operation $e \vee m$ is applied, where m is the indifference factor (equal to 1 if the user does not mind if vegan food is offered, and 0 if the user does mind):

$$r = m + e - me,$$

resulting in $r = 1$ if the user is indifferent or if the user cares and the restaurant also offers this type of food.

4.3.2 Implementation Decisions for Solution 1

This subsection discusses the design choices made for Solution 1, which follows the structure outlined in Section 4.1.1. It covers, among others, the parameter selection, the filtering of relevant POIs sent to the user, and the necessity of sending fake POIs to server A.

4.3.2.1 Parameters Selection

The parameter selection for architecture 1 is shown in Listing 4.1.

```
params.SetMultiplicativeDepth(4);
params.SetSecurityLevel(HEStd_128_classic);
params.SetScalingModSize(50);
params.SetFirstModSize(51);
```

Listing 4.1: Parameters of Solution 1

The multiplicative depth was set to 4, as the architecture involves few homomorphic multiplications. The security level selected was HEStd_128_classic, aligning with OpenFHE's recommended parameters. A ring dimension of 32,768 was automatically chosen by the library to meet the required security level. The scaling modulus and first modulus were set to 50 and 51 bits, respectively, as the design generates minimal noise. Bootstrapping was not enabled. Batch size was kept variable for performance testing, constrained to less than N/2, where N = 32,768.

4.3.2.2 User Parameter-Based Vector Construction Modification

In this architecture, the calculation of all vectors with user parameters is the same as that mentioned in Section 4.3.1.4, except for the calculation of the **Distance Affinity Vector**.

In this architecture, **server B** and **server A** work together to calculate the distance vector, as explained in Phase 3 of the design of Solution 1, explained in Section 4.1.1.

4.3.2.3 Selection of relevant information

In this architecture, since **server A** can decrypt and filter the data, a mechanism has been added to return only the relevant information to the user.

To achieve this, **server A** continuously receives data from **server B**, withholding any POI information until one of two conditions occurs: either the number of POIs reaches the batch size, or **server B** is on its final iteration and must transmit its remaining data.

As detailed in Phase 3 of Solution 1 (see Section 4.1.1), before sending POIs to **server A**, **server B** applies random noise to mask the true location of each POI. Because of this, it is not enough for **server A** to simply store the POIs and send them once the conditions are met. Instead, when **server A** keeps at least one POI from the provided vector, it must notify **server B** that the corresponding noise vector is relevant. This allows **server B** to retain the noise, so when **server A** sends back the filtered POIs, **server B** can associate and remove the noise, ensuring only the filtered POIs are forwarded to the client.

4.3.2.4 Fake points of interest

Because **server A** can decrypt POIs (even if they contain noise) this architecture introduces a privacy risk: **server A** could infer sensitive user information, such as search behaviour or location patterns, especially when few POIs meet the user's restrictions, potentially indicating rural areas or small search radius.

To counteract this, **server B** randomly generates (with a random frequency) and sends, at least once, a vector containing batch-size random components. It also falsifies the vectors used to compute the User Parameter Acceptance value, tricking **server A** into accepting these POIs and thus confusing the analysis to mitigate the privacy risk.

These dummy POIs are later invalidated by **server B** when it subtracts a large value from them (treating it like regular noise). This allows the user to easily identify and discard these points (for instance, by filtering out any POIs with values below -180).

```
vector<uint32_t> levelBudget = { 4, 4 };
vector<uint32_t> bsgsDim = { 0, 0 };
uint32_t levelsForCalc = 12 + 3;
SecretKeyDist secretKeyDist = UNIFORM_TERNARY;
uint32_t multDepth = levelsForCalc + FHECKKSRNS::GetBootstrapDepth(
    levelBudget, secretKeyDist);

params.SetMultiplicativeDepth(multDepth);
params.SetSecurityLevel(HEStd_128_classic);
params.SetScalingModSize(59);
params.SetFirstModSize(60);
params.SetBatchSize(BatchSize);
```

Listing 4.2: Parameters of Solution 2

4.3.3 Implementation Decisions for Solution 2

Throughout this subsection, the decisions taken for the design of Solution 2 will be addressed, which follows the design specified in Section 4.1.2. Topics will be addressed from how the parameters were selected, how the homomorphic comparison between two ciphertexts was addressed, to how the POIs are filtered.

4.3.3.1 Parameters Selection

The parameters selection for this solution differ from the previous one in Section 4.3.2.1, since this solution needs to apply bootstrapping as the number of multiplications is greater than in Solution 1. The parameter selection ends up as in the Listing 4.2.

For the bootstrapping setup, a level budget of 4,4 has been defined, following the library's recommendations for large ring sizes. The baby-step giant-step (BSGS) parameters used in the linear transformation algorithm for encoding and decoding during bootstrapping have been set to {0,0}, allowing OpenFHE to automatically select optimized values on its own. Additionally, a uniform ternary secret key distribution was chosen, since it meets the standard for homomorphic encryption.

Regarding the general parameters, the multiplicative depth was configured to support twelve computation levels plus three additional levels required for periodic bootstrapping. The final multiplicative depth was determined using the function "GetBootstrapDepth()", which returns the multiplication depth needed for bootstrapping with the specified level budget and secret key distribution. A scaling factor of 2^{59} and a first modulus of 2^{60} bits were selected to control noise growth during multiplications, allowing at least 15 multiplications before a bootstrapping is necessary. The batch size remains variable for performance testing, with the only restriction being that it must be less than N/2, where N=2^{17}=131072. This ring dimension also meets the HEStd_128_classic security level (again this parameter has been selected automatically by the library as in Section 4.3.2.1), ensuring compliance with OpenFHE's parameter selection guidelines.

4.3.3.2 Homomorphic comparison

In this architecture, no server can decrypt, so the comparison between two ciphertexts must be performed homomorphically. For this purpose, the following ninth-degree polynomial was used [5]:

$$f(x) = \frac{35}{128}x^9 - \frac{180}{128}x^7 + \frac{378}{128}x^5 - \frac{420}{128}x^3 + \frac{315}{128}x$$

The described polynomial is an approximation of the sign function, which returns:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0, \\ -1 & \text{if } x < 0. \end{cases}$$

This polynomial is designed for normalized inputs in the range $[-1, 1]$, since approximations tend to diverge or lose precision outside this range.

Once this polynomial was defined, Algorithm 1 was implemented, which applies this polynomial approximation to the domain of the CKKS homomorphic scheme. Note that rescaling was added to transform the output in the range $[-1, 1]$ to the range $[0, 1]$ in order to be able to use the output of this comparison as a binary mask.

4.3.3.3 User Parameter-Based Vector Construction normalization

As previously specified in Subsection 4.3.3.2, the x values for the polynomial must be normalized in the range $[-1, 1]$, but the vectors calculated for the acceptance of the user parameters are not normalized, because the quadratic Euclidean distances can be very small values, but they can also be larger values (on the order of values greater than a thousand), so some normalizations have been applied in order to be able to apply the Algorithm 1:

- Distance Affinity Vector: To normalize the quadratic Euclidean distances, the function implemented in OpenFHE called "EvalChebyshevFunction" is used,

Algorithm 1 Homomorphic Comparison of Two Ciphertexts

```
Require: $ciph_1, ciph_2$: ciphertexts of type Ciphertext<DCRTPoly>
Require: $compIterations$: number of comparison iterations
Ensure: $r$: ciphertext equal to 0 if $ciph_1 < ciph_2$, 1 if $ciph_1 > ciph_2$, 0.5 if $ciph_1 = ciph_2$
 1: $bootstr \leftarrow 1$
 2: $precisionAfterFirstIteration \leftarrow 18$
 3: $result \leftarrow \mathsf{EvalSub}(ciph_1, ciph_2)$
 4: for $i = 0$ to $compIterations - 1$ do
 5:     $coeff_9 \leftarrow 35$
 6:     $coeff_7 \leftarrow 180.0$
 7:     $coeff_5 \leftarrow 378.0$
 8:     $coeff_3 \leftarrow -420.0$
 9:     $coeff_1 \leftarrow 315.0$
10:     $denom \leftarrow \frac{1.0}{128.0}$
11:     $x_2 \leftarrow \mathsf{EvalMult}(result, result)$
12:     $x_4 \leftarrow \mathsf{EvalMult}(x_2, x_2)$
13:     $tmp1 \leftarrow \mathsf{MultByInteger}(x_4, coeff_9)$
14:     $tmp2 \leftarrow \mathsf{MultByInteger}(x_2, coeff_7)$
15:     $tmp3 \leftarrow \mathsf{EvalSub}(tmp1, tmp2)$
16:     $term1 \leftarrow \mathsf{EvalAdd}(tmp3, coeff_5)$
17:     $term1 \leftarrow \mathsf{EvalMult}(term1, x_4)$
18:     $term2 \leftarrow \mathsf{EvalAdd}(\mathsf{EvalMult}(coeff_3, x_2), coeff_1)$
19:     $term3 \leftarrow \mathsf{EvalMult}(result, denom)$
20:     $result \leftarrow \mathsf{EvalMult}(\mathsf{EvalAdd}(term1, term2), term3)$
21:     if $bootstr \geq 3$ then
22:         $result \leftarrow \mathsf{EvalBootstrap}(result, 2, precisionAfterFirstIteration)$
23:         $bootstr \leftarrow 1$
24:     else
25:         $bootstr \leftarrow bootstr + 1$
26: $r \leftarrow \mathsf{EvalMult}(0.5, \mathsf{EvalAdd}(1, result))$ return $r$
```

which generates and evaluates a Chebyshev polynomial that approximates a function.

To normalize the distances, the $\tanh(x)$ function is used, defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

which, for an input x, normalizes the value to the range $[-1, 1]$. As stated before, maintaining the sign is important to perform the comparison correctly.

The selection of the $\tanh(x)$ function instead of the alternative function:

$$f(x) = \frac{x}{|x| + 1}$$

is due to the fact that evaluating the second function requires significantly greater depth, since in addition to performing the division by the reciprocal of x, it also needs to evaluate the absolute value $|x|$, adding more complexity.

Additionally, a *pre-normalization* step has been applied in order to limit the input values and prevent excessively large values that could cause the generated Chebyshev polynomial to diverge.

- Average price vector and cuisines affinity: To normalize these values, what has been done is to divide them by a fixed number, and thus scaling the vectors in a range [-1,1].

4.3.3.4 POIs filtering

Once the vectors containing the conditions for determining whether a POI meets the user's constraints have been calculated, a comparison of all these vectors with a vector of all zeros was performed. This allows to compare whether all the components of the vectors are greater than zero. These comparisons are performed in pairs, and once all of them have been completed, the results of all the comparisons are multiplied to generate a mask, which is then multiplied by the POIs and invalidates any data the user is not interested in.

Note that once the mask has been created an additional comparison must be performed with a vector filled with the fixed value 0.5. This is necessary because when comparing values close to 0 but not 0, such as 0.7, the value returned by the compare function is close to 1, but not exactly 1 (it could be 0.9), and if we apply the mask with that value, the POI location would lose precision.

Accordingly, by applying the compare function again on the mask we make sure that the values greater than 0.5 and that should be greater than 0.9 go to 0.99

which does not take away barely any precision from the POI and offers a much more reliable and precise mask.

Nevertheless, when applying the approximate comparison function to certain parameters (especially when comparing the distance constraint) values extremely close to zero may be misclassified, even though they satisfy the constraint. This invalidates the mask for those points, potentially causing missing points in our solution.

4.3.4 Data Source

For data collection in this thesis, the Overpass Turbo [28] application was chosen. This tool serves as a data filter for OpenStreetMaps [27] and was selected due to its simpler interface for querying, downloading, and handling data compared to the main OpenStreetMaps website.

Using this platform, over 80,000 restaurants across Germany were initially retrieved. However, since the computation time scales exponentially with the number of data points, the sample was reduced to 20,000 restaurants to speed up the execution of the tests and obtain results more efficiently.

Chapter 5

Performance Evaluation

Evaluation results for the two architectures were obtained with the data set described in Section 4.3.4 and with a variable Batch Size. The two architectures are analysed separately in each section, as each uses different parameters, allowing for an individual breakdown of their outcomes. All tests were conducted on a server owned by the University of Zaragoza, accessible to all computer science students of this institution. It features a 16-core (32-thread) AMD EPYC 7313P processor, running at a base frequency of 1,500 MHz with a boost up to 3,729 MHz, and has 514 GB of available RAM.

To obtain the times and resources consumed in this chapter, the Linux `time` command (`/usr/bin/time`) is used, which is capable of capturing RAM consumption, as well as processor usage times in user or system mode. In addition, to capture the times between different phases within the program, the chrono library is used, which allows us to capture times with high precision.

The accuracy of the solutions was measured using a Python program, which calculates the average error in meters of the points obtained and identifies the number of missing points produced by the solution.

5.1 Memory Usage

Figure 14 shows the maximum memory consumption recorded during the tests of Solution 1. Memory usage remains essentially constant (around 0.62 GB) even if the batch size grows from 1,024 to 16,384. This stability stems from the moderate ring dimension (2^{15}) and the fact that Solution 1 applies decryption during the filtering in **Server A** operations instead of accumulating intermediate ciphertexts, and does not invoke the bootstrapping operation. As a result, internal data structures, keys

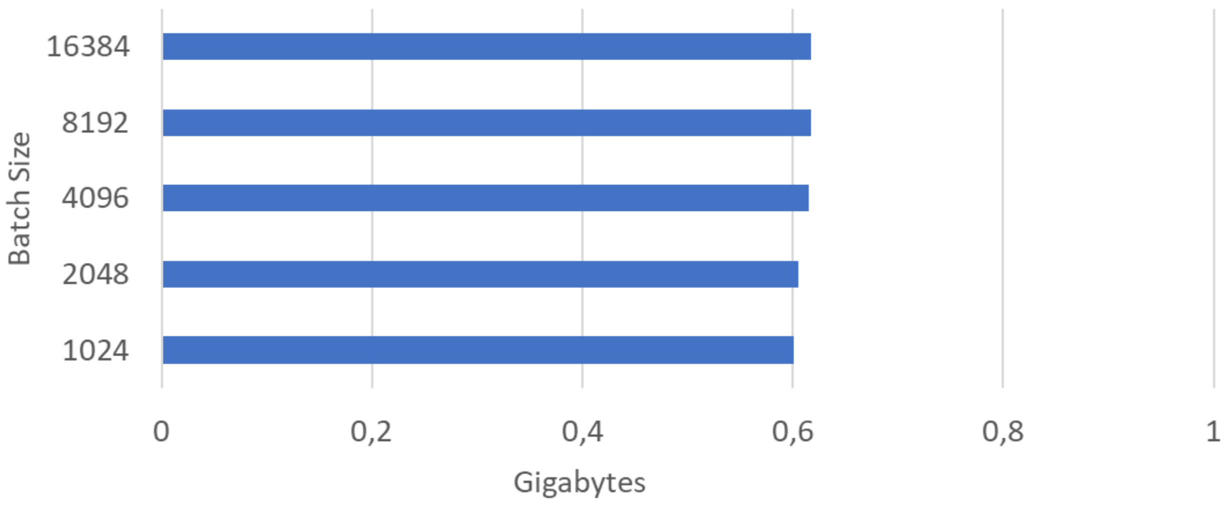


Figure 14: Memory Consumption of Solution 1

and related overhead remain fixed. The slight increase in RAM at larger batch sizes is due to the larger ciphertext objects and is negligible, allowing us to approximate the overall memory usage as constant.

Now, Figure 15 illustrates the peak RAM consumption measured for Solution 2. Memory usage rises with batch size, reaching approximately 91 GB at the largest batch. Unlike Solution 1, RAM consumption is not constant: it grows with batch size primarily because Solution 2 uses a larger ring dimension (2^{15}), generates and retains more intermediate ciphertexts to homomorphically filter points of interest, evaluates a degree-9 polynomial to approximate the sign function, and performs bootstrapping, which requires memory-intensive rotation keys. These factors drive the RAM footprint of Solution 2 to be an order of magnitude higher than that of Solution 1.

5.2 Database Size

Figure 16 plots the total encrypted database size for Solution 1 versus batch size. Because the ring dimension remains fixed at 2^{15}, smaller batch sizes require more ciphertexts to represent the full dataset, producing a maximum storage footprint of 0.8 GB. As batch size increases and packing efficiency improves, the required storage falls off exponentially.

While Figure 17 plots the total encrypted database size for Solution 2 as a function of batch size. Due to the larger ring dimension (2^{17}), the database occupies approximately 25 $\times$ more storage than in Solution 1. As batch size increases,

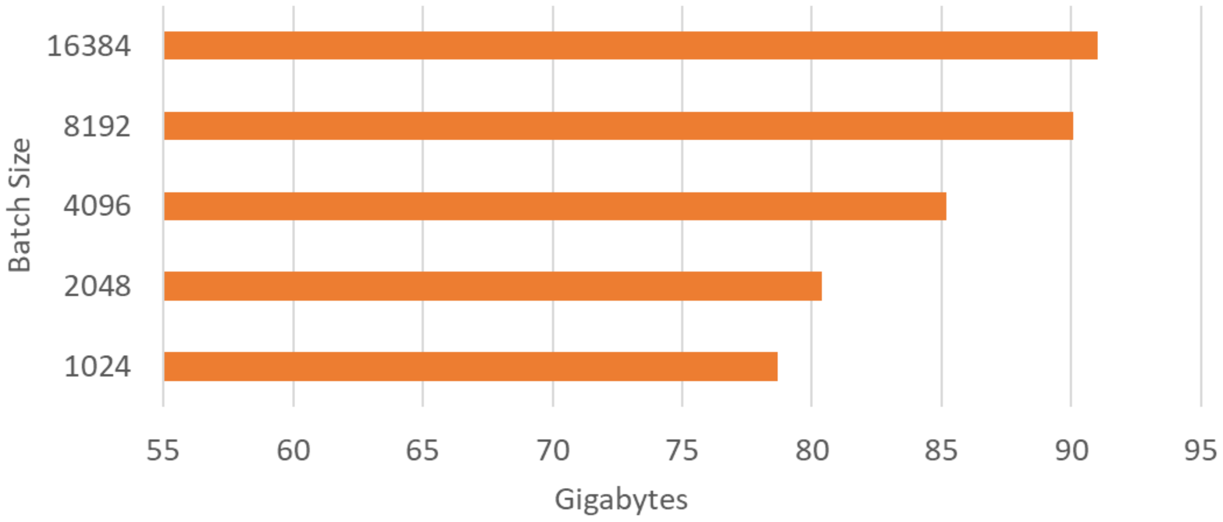


Figure 15: Memory Consumption of Solution 2

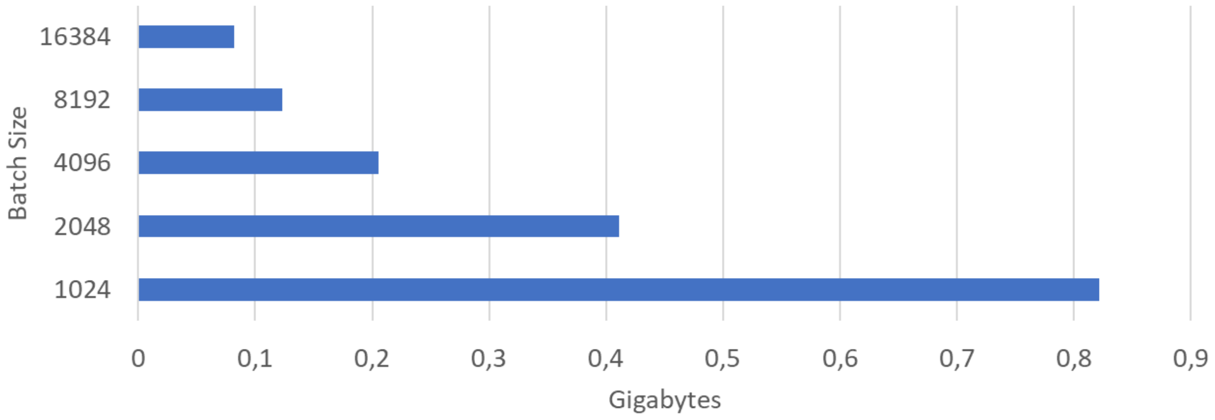


Figure 16: Database Size of Solution 1

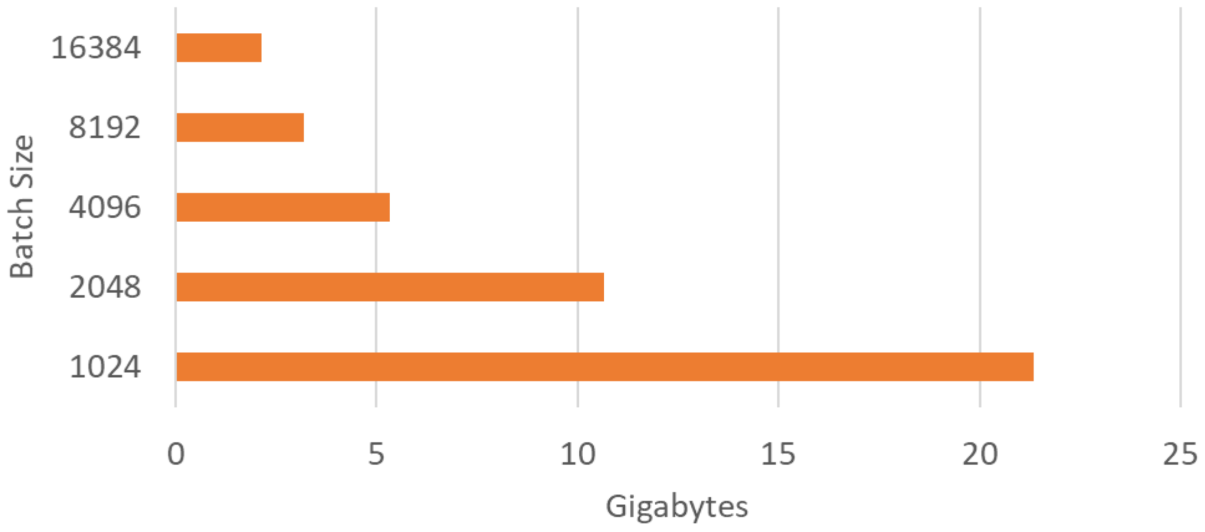


Figure 17: Database Size of Solution 2

the required storage decreases exponentially, again reflecting improved ciphertext packing efficiency at higher batch sizes.

5.3 Time Efficiency

Now Figure 18 plots the time efficiency of Solution 1's main phases: key generation, client-side query composition, database encryption, and the filtering stage executed by **Server B** (in conjunction with **Server A**),for batch sizes ranging from $1,024$ to $16,384$. As batch size increases, the runtimes of database encryption and the filtering phase decrease proportionally to the reduced number of ciphertexts, while key generation and query composition remain constant (they depend only on the fixed ring dimension, 2^{15}, and the number of packed slots). The filtration phase is the one that takes the longest latency, taking roughly 12 seconds at a batch size of $1,024$ and dropping to about 2 seconds at $16,384$.

Figure 19 breaks down total CPU time into user and system time. User time accounts for approximately 86% of CPU cycles, with system time at about 14%. This confirms that the workload is predominantly on homomorphic operations, rather than I/O operations by database access.

While Figure 20 shows the time efficiency of Solution 2. Compared to Solution 1, runtimes increase by approximately 850%. This performance penalty arises from several factors: the ring dimension increase from 2^{15} to 2^{17} (slowing arithmetic operations); no intermediate decryption is performed, so to perform the

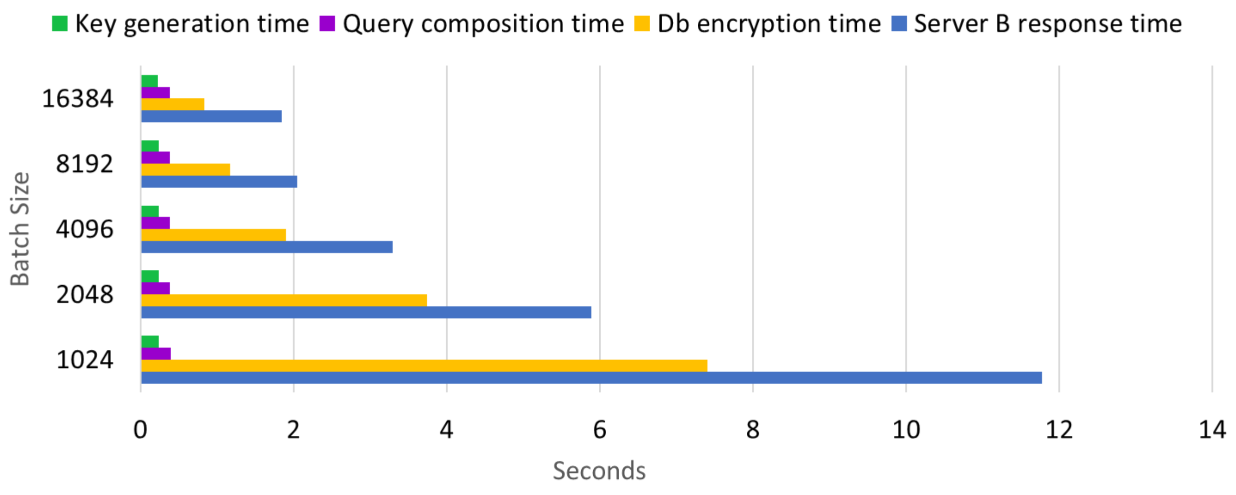


Figure 18: Time Efficiency of Solution 1

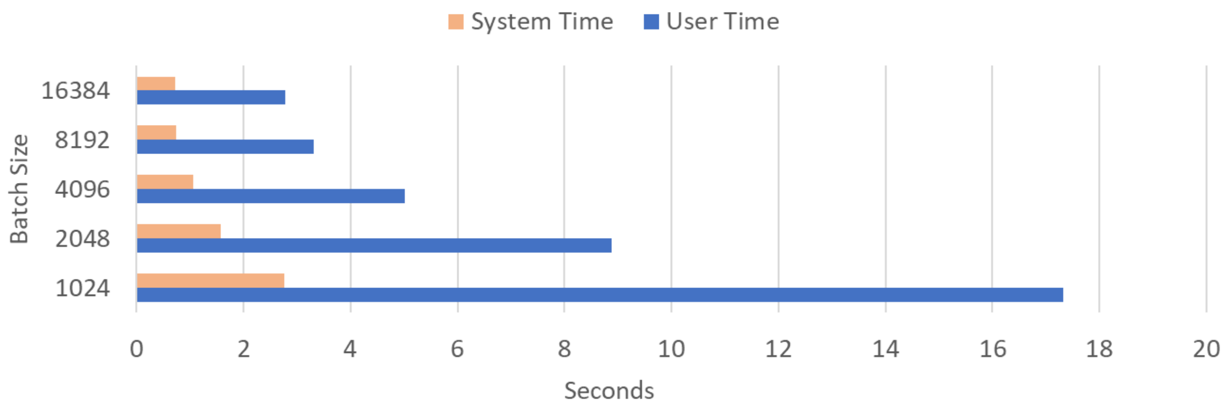


Figure 19: User vs System Time of Solution 1

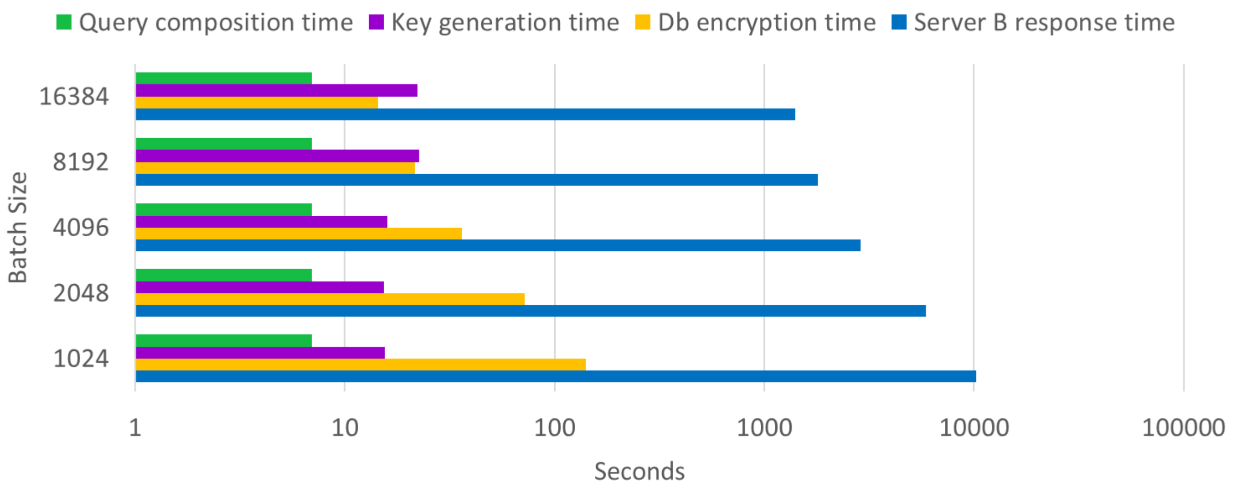


Figure 20: Time Efficiency of Solution 2

comparison, we need to apply the nine-degree polynomial; each comparison invokes bootstrapping; and more intermediate ciphertexts are retained. Consequently, at a batch size of $1,024$, **Server B**'s filtering stage takes nearly $10,000$ seconds and database encryption consumes about 140 seconds, while key generation and query composition remain essentially constant across batch sizes.

As batch size increases, runtimes improve exponentially: at a batch size of $16,384$, filtering time falls to roughly $1,400$ seconds, and encryption time to around 15 seconds, achieving a reduction of the $\sim 86\%$ relative to the $1,024$ case.

Figure 21 decomposes total CPU time into user and system components. Both metrics exceed those of Solution 1, reflecting increased I/O overhead due to a larger encrypted database. However, user time (homomorphic computations) still dominates overall execution.

5.4 Precision Achieved

Regarding the accuracy obtained from the two different architectures, various metrics were collected, including the average error measured in meters and percentage terms, the number of missing points, and the overall accuracy achieved by both solutions.

The collected data is shown in Table 5.1, where it can be observed that both solutions yield the same precision error, with an average deviation of 4.23 meters from the original point, or a variation of 0.42% compared to the plaintext point. The

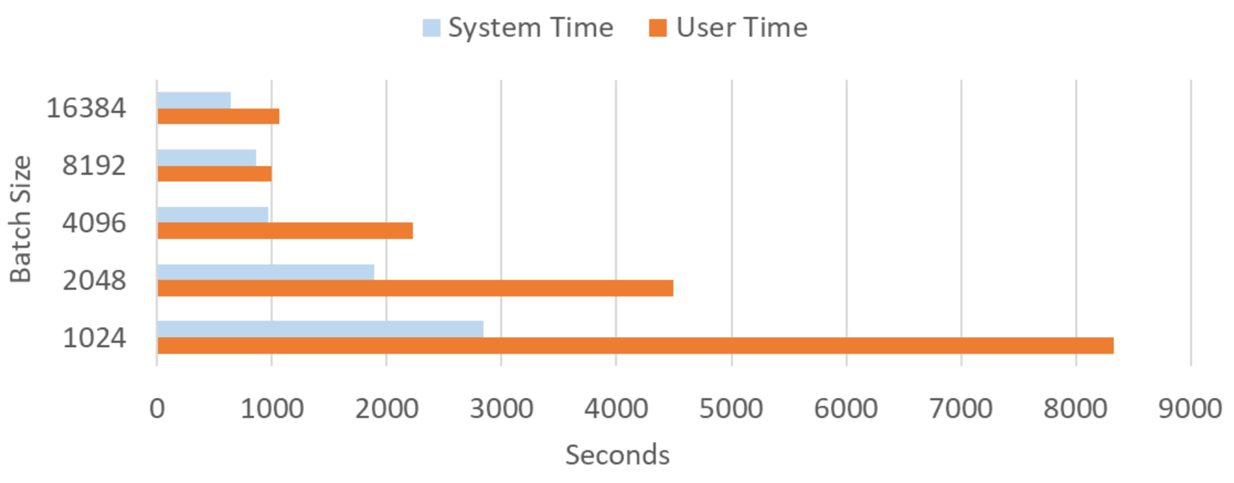


Figure 21: User vs System Time of Solution 2

Metric	Solution 1: Two cloud servers	Solution 2: One Cloud Server
Mean error	0.42% (4.23 meters)	0.42% (4.23 meters)
Missing points	0%	8.42%
Overall accuracy	100%	91.57%

Table 5.1: Accuracy comparison between the two solutions

difference arises when comparing the points that both solutions fail to detect due to possible approximation errors or edge cases related to user-defined parameters. Since CKKS uses approximate arithmetic, it may interfere with these results. The first difference observed is that in Solution 1, no missing points were detected. This is due to the use of very low multiplicative depth and the application of decryption to filter data, which avoids noise accumulation and loss of precision. In contrast, the second solution uses a comparison method based on the approximation of the sign function, as explained in subsection 4.3.3.2, which can cause edge cases to be evaluated as negative. Consequently, this results in a zero during mask generation, leading to the incorrect exclusion of a restaurant that should not have been discarded.

Overall, we can say that the accuracy of Solution 1 is approximately 100%, while the accuracy of the second solution is 91.57%, which is still a very good value considering the multiplicative depth of the circuit and the bootstrapping operation.

Chapter 6

Conclusion

This work started from a clear problem: providing location-based services without exposing the user's exact position or the query parameters. The goal was to show that it is possible to retrieve points of interest while preserving the confidentiality of both the database and the requests, with zero linkability and zero observability, and a minimum security level of 128 bits.

To achieve this, we designed two architectures. The first uses two servers that never cooperate. Server B masks all data so that Server A cannot access any relevant information. Server A then decrypts and filters the points of interest based on the masked values it receives from Server B. The second uses a single server with no decryption capability, so the database remains fully encrypted and comparisons run on approximate arithmetic within the CKKS scheme. This boosts privacy but increases computation and memory load. Both designs meet the functional requirements GR1 and GR2, and the privacy requirements PR1, PR2, and PR3 defined at the start of this thesis. It is worth noting that these architectures and their protocols were designed specifically for the proposed scenario, but to adapt this solution for another scenario, it might require significant adaptations, since homomorphic solutions usually require very specific configurations.

We ran experiments on 20,000 restaurants from OpenStreetMap, using batch sizes from $1,024$ to $16,384$. In terms of accuracy, both solutions show a mean error of 4.23 meters (0.42%) compared to the plaintext coordinate. However, the second solution omits 8.42% of points because of the approximate comparison, yielding an overall accuracy of 91.57% versus 100% for the first. In memory, Solution 1 stabilises around 0.62 GB because it offloads filtering to an actor that can decrypt and work on plaintext. The single-server solution climbs to 91 GB at the largest batch, due to a larger ring dimension, stored intermediate ciphertexts, and rotation keys needed for bootstrapping. Regarding performance times, Solution 1 filtering drops from twelve to two seconds as the batch grows, while Solution 2 goes from nearly $10,000$ seconds down to $1,400$ seconds at the same scale, a worst-case change of 850%, explained by the lack of decryption and the cost of bootstrapping.

This work reveals three main limitations. First, the dataset is static, which means the prototype does not support dynamic inserts or deletes and needs re-encryption whenever the database is updated. Second, it relies on OpenFHE, whose official interfaces are limited to C++, Rust, and Python; porting the system to mobile devices would require extra native bindings. Third, scaling the system to many concurrent users introduces an exponential rise in RAM usage because each extra user adds more ciphertexts in memory. If the space used by ciphertexts is not optimised, the memory footprint can quickly exceed the capacity of a single machine. In addition, processing several ciphertexts at once can extend execution time far beyond practical limits. Running the service at commercial scale would therefore demand very large and powerful infrastructures, making this scalability issue a critical constraint on real-world deployment.

These findings suggest several improvement paths: researching compressed rotation keys, adding encrypted incremental updates to enable real-time data, querying a subset of the database for faster execution than querying the entire database, and exploring hardware accelerators to reduce filtering times when the POI database grows to millions of records, developing ciphertext-compaction and reuse strategies such as packing several user queries in a single homomorphic vector and designing a distributed microservice architecture with load balancing and ciphertext sharding so that multiple machines share the computational load.

In summary, this thesis demonstrates that using CKKS enables private location queries with a sub-5-meter average error from the original point to the resulting one and classical 128-bit security. Choosing between the two architectures means balancing accuracy and resource use: the two-server option achieves full precision with moderate memory and time requirements, while the single-server option maximises confidentiality at the cost of higher computational demands and a slight drop in coverage. Yet both approaches face a major scaling challenge: without more efficient ciphertext storage and distributed processing, memory and compute requirements grow rapidly with the number of users, limiting commercial deployment. The proposed improvements, especially those targeting ciphertext optimization and distributed execution, outline a concrete path to move the prototype toward production environments where the database evolves continuously, must support many simultaneous clients, and operates at industrial scale.

References

[1] Martin R Albrecht, Rachel Player, and Sam Scott. “On the concrete hardness of learning with errors”. In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203. DOI: 10.1515/jmc-2015-0016.

[2] Miguel E. Andrés et al. “Geo-indistinguishability: Differential Privacy for Location-Based Systems”. In: *Proceedings of the 20th ACM Conference on Computer and Communications Security*. 2013. DOI: 10.1145/2508859.2516735. URL: https://doi.org/10.1145/2508859.2516735.

[3] Beresford et al. “Mix zones: User privacy in location-aware services”. In: *2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2004. URL: https://www.cl.cam.ac.uk/~fms27/papers/2004-BeresfordSta-mix.pdf.

[4] Zvika Brakerski and Vinod Vaikuntanathan. “Leveled fully homomorphic encryption without bootstrapping”. In: *Innovations in Theoretical Computer Science (ITCS)*. ACM. 2014, pp. 309–325. DOI: 10.1145/2554797.2554798.

[5] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. “Efficient Homomorphic Comparison Methods with Optimal Complexity”. In: *Advances in Cryptology – ASIACRYPT 2020*. Springer International Publishing, 2020, pp. 221–256. ISBN: 978-3-030-64834-3. DOI: 10.1007/978-3-030-64834-3_8.

[6] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: 2017. DOI: 10.1007/978-3-319-70694-8_15. URL: https://www.iacr.org/archive/asiacrypt2017/106240294/106240294.pdf.

[7] Chor et al. “Private Information Retrieval”. In: *36th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 1995. DOI: 10.1145/293347.293350. URL: https://www.researchgate.net/publication/220432020_Private_Information_Retrieval.

[8] Cynthia Dwork et al. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: Springer, 2006. DOI: 10.1007/11681878_14. URL: https://doi.org/10.1007/11681878_14.

[9] The Editors of Encyclopaedia Britannica. *Latitude and longitude explained.* `https://www.britannica.com/science/latitude`. 2025.

[10] Endeavour Sailing. *Latitude and Longitude.* `https://www.endeavour-sailing.co.uk/training-information/navigation/latitude-and-longitude`.

[11] Faneela et al. *Cross-Platform Benchmarking of the FHE Libraries: Novel Insights into SEAL and OpenFHE.* 2025. URL: `https://eprint.iacr.org/2025/473`.

[12] Gentry and Craig. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the 41st annual ACM symposium on Theory of computing (STOC).* ACM. 2009. DOI: `10.1145/1536414.1536440`. URL: `https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf`.

[13] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM* 43.3 (1996), pp. 431–473. DOI: `10.1145/233551.233553`. URL: `https://doi.org/10.1145/233551.233553`.

[14] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. "New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks". In: 2022. URL: `https://eprint.iacr.org/2022/425`.

[15] Gruteser et al. "Anonymous usage of location-based services through spatial and temporal cloaking". In: *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys).* 2003. DOI: `10.1145/1066116.1189037`. URL: `https://dl.acm.org/doi/10.1145/1066116.1189037`.

[16] Lulu Han et al. "Fully privacy-preserving location recommendation in outsourced environments". In: *Ad Hoc Networks* 141 (2023), p. 103077. DOI: `10.1016/j.adhoc.2022.103077`. URL: `https://www.sciencedirect.com/science/article/pii/S1570870522002499`.

[17] Alexandra Henzinger et al. "One Server for the Price of Two: Simple and Fast Single-Server PIR". In: *Proceedings of the 32nd USENIX Security Symposium* (2023). URL: `https://eprint.iacr.org/2022/949.pdf`.

[18] Lei Jiang and Lei Ju. "FHEBench: Benchmarking Fully Homomorphic Encryption Schemes". In: 2022. URL: `https://arxiv.org/abs/2203.00728`.

[19] Jiang et al. "Location Privacy-Preserving Mechanisms in Location-Based Services: A Comprehensive Survey". In: (2021). DOI: `10.1145/3423165`. URL: `https://www.researchgate.net/publication/348179398_Location_Privacy-preserving_Mechanisms_in_Location-based_Services_A_Comprehensive_Survey`.

[20] Kido et al. “Protection of location privacy using dummies for location-based services”. In: *21st International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2005. DOI: 10.1109/icde.2005.269. URL: https://ieeexplore.ieee.org/document/1647865.

[21] Kushilevitz et al. “One-server private information retrieval”. In: *Proceedings 38th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 1997. URL: https://web.cs.ucla.edu/~rafail/PUBLIC/34.pdf.

[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “Ideal lattices in cryptography”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2010, pp. 1–23. DOI: 10.1007/978-3-642-15369-3_1.

[23] Eny Maria et al. “Measure distance locating nearest public facilities using Haversine and Euclidean Methods”. In: (Feb. 2020). DOI: 10.1088/1742-6596/1450/1/012080. URL: https://ui.adsabs.harvard.edu/abs/2020JPhCS1450a2080M/abstract.

[24] *Microsoft SEAL*. https://github.com/Microsoft/SEAL. Jan. 2023.

[25] Miers et al. “Updatable Zero-Knowledge Databases”. In: *ASIACRYPT*. Springer. 2005. URL: https://link.springer.com/chapter/10.1007/11593447_10.

[26] OpenFHE Project. *OpenFHE: Open-Source Fully Homomorphic Encryption Library*. https://openfhe.org/. 2025.

[27] OpenStreetMap contributors. *OpenStreetMap map data for thousands of websites, mobile apps, and hardware devices*. https://www.openstreetmap.org/about. 2025.

[28] *Overpass Turbo*. https://overpass-turbo.eu/. 2025.

[29] Pascal Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Advances in Cryptology — EUROCRYPT ’99*. Ed. by Jacques Stern. Springer Berlin Heidelberg, 1999. URL: https://link.springer.com/chapter/10.1007/3-540-48910-X_16#citeas.

[30] Yuriy Polyakov. *Homomorphic Encryption for OpenFHE Users: Tutorial with Applications Part 1 – Introduction to Homomorphic Encryption*. https://openfhe.org/portfolio-item/homomorphic-encryption-for-palisade-users/. [Online; accessed 31-May-2025]. Aug. 2020.

[31] Popa et al. “CryptDB: Protecting confidentiality with encrypted query processing”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM. 2011. DOI: 10.1145/2043556.2043566. URL: https://dl.acm.org/doi/10.1145/2043556.2043566.

[32] Taimur Rahman et al. “Benchmarking Fully Homomorphic Encryption Libraries in IoT Devices”. In: Association for Computing Machinery, 2025. URL: https://doi.org/10.1145/3704522.3704546.

[33] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40. DOI: 10.1145/1568318.1568324.

[34] Ronald L. Rivest and Michael L. Dertouzos. “ON DATA BANKS AND PRIVACY HOMOMORPHISMS”. In: 1978. URL: https://api.semanticscholar.org/CorpusID:6905087.

[35] Paul Schmitt et al. “Oblivious DNS: Practical Privacy for DNS Queries”. In: (2019). DOI: 10.2478/popets-2019-0028.

[36] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. *Is Homomorphic Encryption the Holy Grail for Database Queries on Encrypted Data?* Tech. rep. University of California, Santa Barbara, 2012. URL: https://cs.ucsb.edu/sites/default/files/documents/2012-01.pdf.

List of Figures

List of Tables

Listings