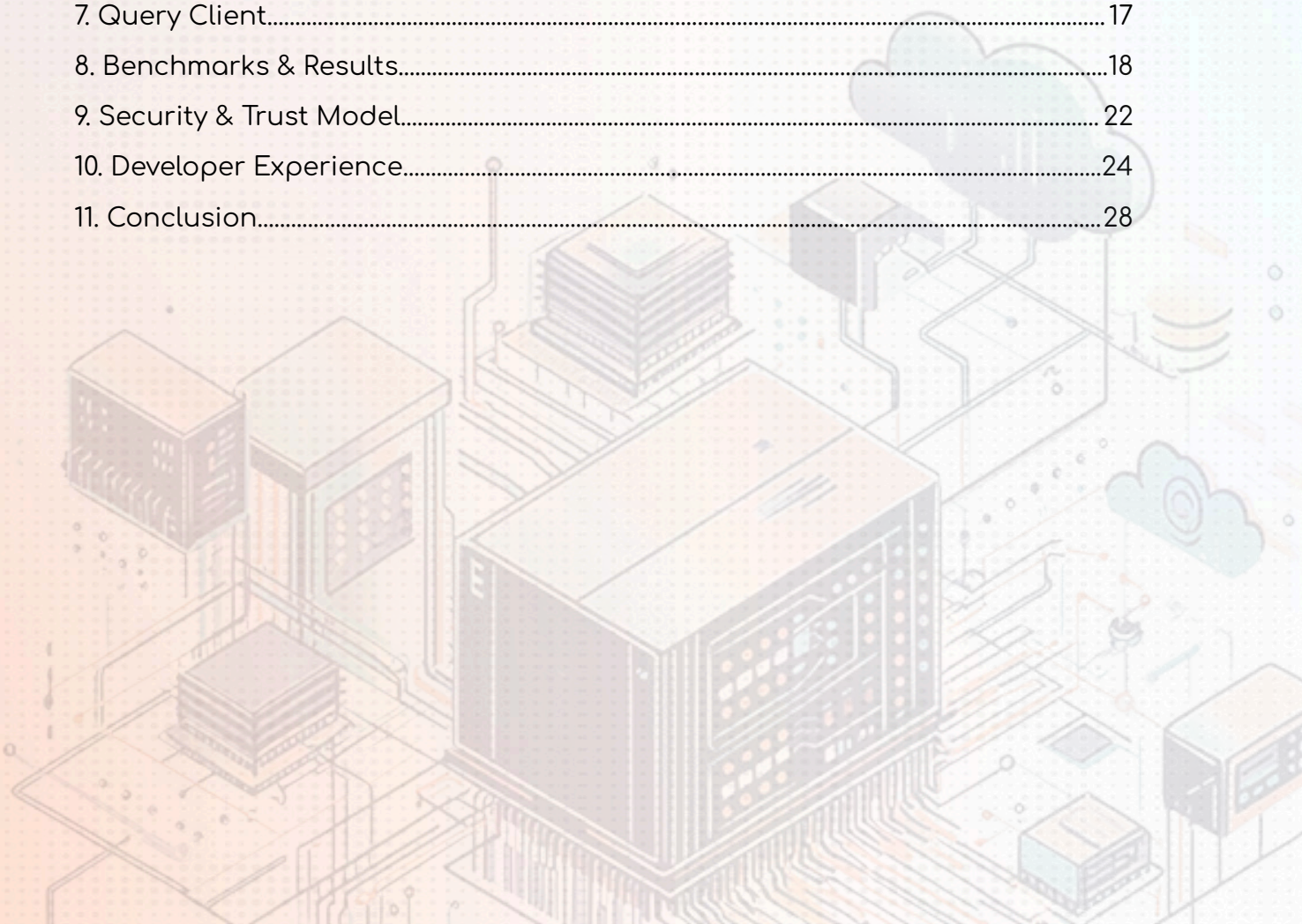
LatticaAI

Technical Whitepaper

Encrypted AI Inference with Fully Homomorphic Encryption



1. Introduction

Artificial Intelligence is becoming a critical enabler across industries such as healthcare, finance, and government. Yet, adoption is increasingly constrained by data privacy, compliance requirements, and trust. Enterprises often cannot leverage their most valuable datasets, such as patient records, financial transactions, or biometric identifiers, because sharing this data with third-party AI providers creates regulatory and reputational risk.

Fully Homomorphic Encryption (FHE) offers a breakthrough: it allows computation directly on encrypted data. With FHE, sensitive inputs never need to be decrypted by the service provider. A model can run in the cloud, yet the server never sees the raw data, and only the data owner can decrypt the result. In principle, FHE solves the privacy problem at its root.

The challenge is that FHE, while mathematically powerful, has not been practical in deployment. A naïve implementation is orders of magnitude slower than plaintext computation, making even simple models prohibitively expensive. Existing libraries were designed as research tools: they expose low-level cryptographic primitives but do not integrate with modern ML frameworks or production pipelines. And although specialized hardware for FHE is emerging, there has been no software layer that bridges cryptography with accelerators, leaving a gap between theoretical capability and usable performance.

Lattica exists to close this gap. Our expertise lies in re-engineering FHE algorithms so they run efficiently on accelerators such as GPUs, and in the future, custom chips. We restructure core FHE primitives such as polynomial multiplication in the CRT basis, modular arithmetic and ciphertext slot rotations, into forms that can fully exploit parallel hardware. We implement these at the CUDA and Torch (C++) level, with PyTorch bindings layered on top to give developers a familiar API.

The result is a platform where ML engineers deploy models for encrypted inference without cryptographic expertise, while hardware vendors integrate their accelerators without building a full ecosystem. Our work makes FHE practical, performant, and production-ready.

This white paper explains the technology behind Lattica: how we implement FHE using tensor abstractions (via Torch), how we accelerate it on GPUs and extend to custom hardware, and how we provide a secure, developer-friendly platform for encrypted inference.

2. Background

What is FHE?

Fully Homomorphic Encryption (FHE) is a cryptographic technique that allows data to be processed while it remains encrypted. A data owner can encrypt sensitive information and send it to a compute provider (for example, a cloud service running an AI model). The compute provider performs the requested computation directly on the encrypted data, without ever decrypting it. When the result is returned, only the data owner, who holds the secret key, can decrypt it to reveal the correct output.

Mathematically, if $Enc(m)$ denotes the encryption of message m, then for any function f:

$$Dec(\hat{f}(Enc(m))) = f(m)$$

where $\hat{f}$ is the homomorphic equivalent of f.

This property enables powerful new applications: secure outsourcing of computation, federated analytics, and AI inference without exposing sensitive inputs.

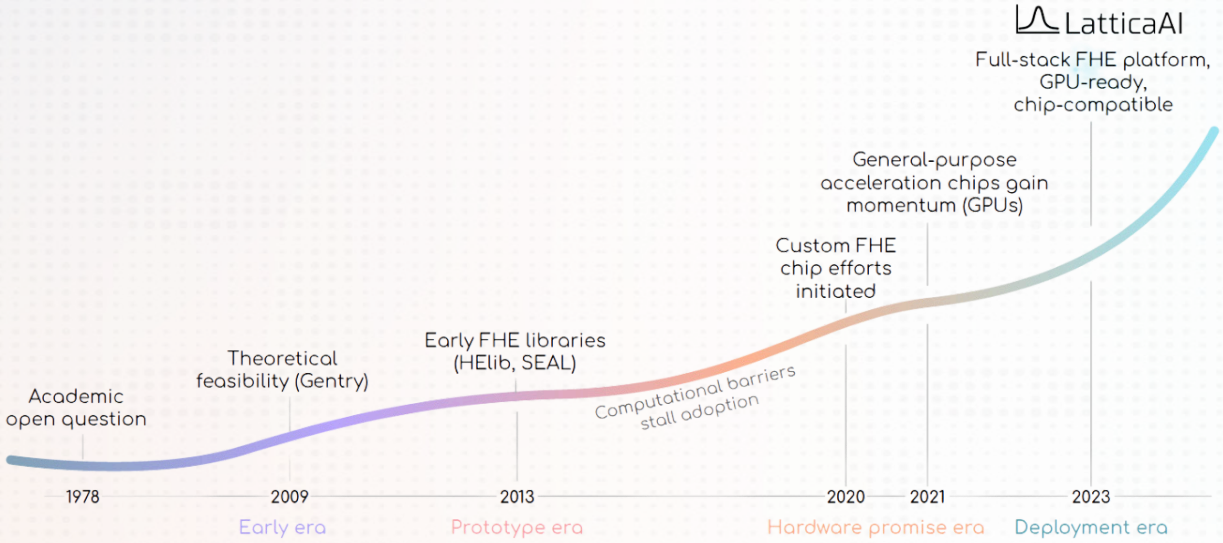
Why Hasn't FHE Been Adopted Widely?

The promise of FHE has been clear since the first schemes were introduced, but adoption has lagged due to three hurdles:

1. **Performance overhead:** Naïve FHE inference can be millions of times slower than plaintext execution. Historically, running even a simple logistic regression under FHE took hours.

2. **Usability:** Libraries such as SEAL, HElib, OpenFHE, and Lattigo were designed as research tools. They expose low-level cryptographic primitives, not developer-friendly APIs for ML engineers. They typically operate as monolithic libraries that tightly couple client and server roles, and do not fit naturally into production ML pipelines.

3. **Lack of hardware integration:** FHE is extremely compute-bound, yet most implementations are CPU-only and not designed to leverage accelerators. Specialized FHE hardware is emerging, but without an abstraction layer, integration is difficult and fragmented.



Evolution of Fully Homomorphic Encryption

Alternatives and Their Limits

Other privacy-preserving technologies have gained adoption but fall short for encrypted inference:

- Differential Privacy: Effective for statistical analysis and training, but adds noise that makes it unsuitable for per-user predictions.

- Trusted Execution Environments (TEEs): Provide near-native performance but require full trust in chip vendors and cloud providers, and have known vulnerabilities. In addition, TEEs tightly couple computation to specific hardware, making it difficult to switch between accelerators, for example, to reduce cost by using cheaper hardware or to scale performance by adopting stronger hardware.

- Secure Multi-Party Computation (SMPC): Introduces high latency for inference due to interactive protocols.

FHE remains the only method that provides strong cryptographic guarantees for per-query inference without requiring trust in infrastructure providers. The challenge is making it fast and usable.

3. Lattica Architecture Overview

Lattica's platform is designed from the ground up to make encrypted inference practical. The architecture has five layers:

Client–Server Split

We separate roles cleanly between the client and the server:

- **Client:** Responsible for key management, input encryption, and result decryption. This can run in lightweight environments such as a mobile app, browser, or enterprise client system. Keys never leave the client environment.

- **Server (Lattica Cloud or on-prem deployment)**: Executes the heavy homomorphic computations on encrypted data. The server only ever processes ciphertexts and never sees raw data or keys.

This split minimizes trust assumptions: the client retains sole control over keys, while the server only processes ciphertexts.

Core Expertise: Cryptography Re-Engineered for Acceleration

The true engine of Lattica's technology is our re-implementation of FHE primitives to exploit modern accelerators. Standard FHE libraries are CPU-bound and monolithic. We redesign the algorithms themselves, e.g. polynomial arithmetic, modular arithmetic, ciphertext rotations and key-switching, so they can be executed in parallel and mapped efficiently onto GPUs today and specialized hardware tomorrow.

Examples of this approach include:

- **Optimized NTT (Number Theoretic Transform) kernels** using CUDA.
- **Batched modular arithmetic** routines aligned with GPU execution patterns.
- Scheduling strategies that minimize memory transfers between CPU and GPU.

This cross-disciplinary expertise, spanning cryptography, GPU kernel programming, and ML systems, is our core technical moat.

HEAL: Abstraction as an Enabler

To make our optimized primitives portable across accelerators, we created the **Homomorphic Encryption Abstraction Layer (HEAL)**.

HEAL defines a lean interface of tensor-level functions that allow hardware vendors to integrate their accelerators into the Lattica ecosystem without re-implementing the entire software stack.

To support integration, we provide a suite of tools alongside HEAL:

- A **standalone runtime** to bring up and test implementations.
- **Unit tests** to validate correctness against reference implementations.

- **Example computation transcripts** drawn from real encrypted inference workloads.

This ensures hardware vendors can validate and optimize their designs quickly, while benefiting from immediate compatibility with the broader Lattica platform.

Layered System: CUDA → Torch → PyTorch

The Lattica platform is built in layers that balance low-level efficiency with high-level usability:

- **CUDA kernels:** Hand-optimized routines for NTTs, modular arithmetic, and other custom FHE transforms.

- **Torch C++ backend:** Manages device memory, orchestrates tensor operations, and bridges CUDA kernels with higher-level APIs.

- **PyTorch bindings**: Expose the system through familiar ML abstractions, allowing developers to integrate encrypted inference with minimal changes to their existing workflows.

For the ML developer, working with Lattica feels like working with PyTorch. Under the hood, however, computations are executed through deeply optimized CUDA/Torch implementations tailored for FHE.

Model Adaptation Layer for ML Inference

Beyond raw homomorphic operations, Lattica includes a **model adaptation layer** specifically designed for ML inference.

When an AI provider uploads a trained model, our platform performs pre-processing on the architecture and weights to adapt them for encrypted execution. This may include:

- Transforming nonlinear layers into FHE-compatible approximations.

- Re-structuring computation graphs to reduce ciphertext depth and rotation overhead.
- Pre-computing constants and packing weights to minimize runtime cost.

All transformations that can be applied ahead of time are performed in this adaptation stage. As a result, the online phase of encrypted inference is reduced to a streamlined sequence of tensor operations, minimizing latency when queries arrive in real time.

4. Tensor Programming Model

Why Tensors Matter

Modern accelerators like GPUs and TPUs are designed around tensors, which are multi-dimensional arrays that can be processed in parallel. A key advantage of tensors is that they allow developers to instruct the hardware to operate over different subsets of dimensions. For example, summing along one axis, multiplying elementwise across another, or collapsing multiple axes into a single one.

Another powerful property of tensors is that they separate the logical data structure from its physical memory layout. Through the use of strides, tensors define how to step through memory along each dimension, enabling different shapes or views of the same data without copying. This makes them especially well suited for accelerators, where memory movement is often more expensive than arithmetic.

For acceleration hardware, tensors provide three critical advantages:

- **Parallelism:** Each dimension can be mapped onto thousands of compute threads.
- **Locality:** Stride-based views allow algorithms to reuse data efficiently without reshaping or copying.

- **Decoupling:** Tensors separate the algorithmic structure of computation from the details of how it is executed on hardware, allowing the same operation to run efficiently across different accelerators.

One of Lattica's core innovations is to reformulate both ciphertexts and algorithms in tensor form. This makes encrypted computation align naturally with accelerator hardware, without requiring algorithm redesign for each backend.

Tensor Representation of Ciphertexts

In FHE schemes such as CKKS and BGV, ciphertexts are pairs of polynomials in the ring

$$R_q = \mathbb{Z}_q[x]/(x^N + 1),$$

with coefficients represented modulo a product of primes $q = \prod q_i$ (the modulus chain). Using the Chinese Remainder Theorem (CRT), each coefficient is represented across these primes simultaneously.

This algebraic structure induces a natural multi-dimensional tensor layout. Typical dimensions include:

- Polynomial degree: indexes the N coefficients of the polynomial.
- Modulus chain: indexes primes in the CRT representation.
- Ciphertext components: two or three components per ciphertext.
- Batch: multiple ciphertexts processed together.

The representation is flexible: additional dimensions can be introduced as needed. For example, gadget decomposition (used in key switching) introduces a dimension for the decomposition basis. Importantly, this same representation also extends naturally to plaintexts: vectors, images, and multi-dimensional feature maps can be packed into tensor slots, so the model's native data layout aligns with the encrypted one.

Exploiting the Tensor Programming Model in Lattica

We don't just represent ciphertexts as tensors, we use this representation as the programming model for building efficient encrypted inference. This has three key consequences:

- **Algorithmic expressiveness:** Complex cryptographic concepts such as homomorphic matrix multiplication, polynomial approximations and bootstrapping can be described naturally in tensor-based programming. Once written this way, they map seamlessly to accelerators, without requiring cryptographers to design hardware-specific variants.

- **Low-level optimization:** At the same time, the tensor representation exposes clean hooks for hand-tuned kernels. We can implement specialized CUDA routines for NTTs, modular arithmetic, or bit decomposition while keeping the higher-level algorithm unchanged. This separation means we can optimize close to the hardware without entangling algorithm design.

- **Hardware agnosticism:** Because tensors capture the parallelism potential of FHE algorithms, our implementation is not tied to any one accelerator. Improvements in GPUs, TPUs, or future FHE-specific chips can be exploited automatically, since the tensor-based language allows us to map operations to whatever parallel resources the hardware provides.

In short, the tensor representation lets us bridge three worlds at once: cryptographic algorithms, accelerator-level optimizations, and hardware portability.

5. GPU Acceleration

Why GPUs for FHE

Fully Homomorphic Encryption is dominated by arithmetic-heavy primitives such as polynomial addition, multiplication, modular reduction, and ciphertext rotations. Once ciphertexts are expressed as tensors, the natural next step is to run them on hardware designed for tensor workloads. GPUs are a strong first platform because they combine:

- **Massive parallelism**: Tens of thousands of lightweight threads allow each coefficient × modulus pair to be updated simultaneously.

- **Large RAM capacity**: FHE ciphertexts are big objects (polynomials with tens of thousands of coefficients, represented across dozens of primes). GPUs provide relatively large on-device memory, making them suitable for storing these expanded ciphertexts and keeping them resident throughout inference.

- **Mature ecosystem**: CUDA, Torch (C++), and PyTorch offer stable toolchains for kernel development, runtime orchestration, and user-facing APIs.

This makes GPUs the first practical accelerator for FHE workloads: widely available, cost-efficient, and capable of handling ciphertexts at scale.

Elementwise Modular Arithmetic

In the CRT representation, nearly all FHE operations reduce to **elementwise modular arithmetic** over the polynomial × modulus chain tensor. The dominant computation is repeatedly applying

$$c_i = (a_i \cdot b_i) \bmod q \quad \text{or} \quad c_i = (a_i + b_i) \bmod q$$

for thousands of coefficients across multiple primes.

The tensor representation allows us to focus optimization effort precisely on this bottleneck. For example:

- We implement custom CUDA kernels with Barrett reduction, avoiding slow integer division in modular multiplication.

- Unlike PyTorch or TensorFlow, which are optimized for floating-point math, our kernels are designed for modular integer arithmetic. If implemented naïvely in Torch, computing `a * b % q` with 32-bit operands forces promotion to 64-bit tensors for intermediate results. This doubles memory usage, meaning only ~50% of GPU RAM can be used to store ciphertexts.

  By contrast, our kernels allocate intermediate high-precision registers inside the GPU thread, so only the final reduced results are written back. This allows us to use up to ~85% of GPU memory for ciphertext storage, dramatically improving capacity and throughput.

Optimized NTT Kernels

Polynomial multiplication in FHE is carried out using the **Number Theoretic Transform (NTT)**, which converts convolution in coefficient space into elementwise multiplication in the NTT domain. For large ciphertext degrees ($N = 2^{12}$ – 2^{16}), this is the single most expensive primitive in the pipeline.

Naïve GPU implementations of NTT perform poorly because they:

- Require frequent global memory shuffles, moving coefficients back and forth across the GPU.

- Introduce non-coalesced memory accesses, where threads read/write scattered addresses.

- Allocate temporary buffers for each stage, increasing memory overhead and reducing effective GPU RAM for ciphertexts.

Our CUDA NTT kernels are designed to minimize these bottlenecks:

- **Shared memory tiling**: At each butterfly stage, coefficients are loaded into on-chip shared memory, reducing global memory traffic by an order of magnitude.

- **Memory coalescing**: Global memory access patterns are carefully aligned so that consecutive threads access consecutive coefficients, maximizing throughput.

- **In-place transformations**: Intermediate results are written back into the same buffer, avoiding allocation of temporary arrays and preserving GPU RAM for ciphertext storage.

With these optimizations, NTT execution scales efficiently even for very large polynomials. For example, an $N = 2^{16}$ transform that takes tens of milliseconds on CPU can be executed in just a fraction of a millisecond on a single GPU.

Custom Kernels for FHE Primitives

While elementwise modular arithmetic and NTTs dominate the runtime, certain cryptographic subroutines are uniquely challenging in FHE and require specialized kernels. Chief among these is key switching, which enables ciphertexts to be transformed after multiplications and rotations.

A central step in key switching is bit decomposition of coefficients, typically across a gadget basis. Naïve implementations first reconstruct coefficients in $\mathbb{Z}_q$ and then decompose them, which is both memory-heavy and inefficient on GPUs.

Our custom kernel performs bit decomposition directly in the CRT domain using techniques borrowed from Garner's algorithm for mixed-radix reconstruction.

We also implemented custom modular gadget multiplication kernels, tuned to GPU integer arithmetic pipelines, that consume the decomposed representation efficiently.

Scheduling and Memory Optimizations

Fast kernels alone are not enough. FHE evaluation involves long sequences of dependent operations, and poor orchestration can easily erase raw kernel speedups. Lattica's runtime applies several scheduling and memory strategies to keep execution efficient end-to-end:

- **Fused execution**: Sequences of dependent operations (e.g., *decompose → multiply → accumulate*) are collapsed into single kernel launches, reducing launch overhead and eliminating unnecessary global memory writes.

- **On-GPU residency**: Ciphertexts remain in device memory for the entire inference pipeline, avoiding host–device transfers that would otherwise dominate latency.

- **Memory-aware scheduling**: Intermediate tensors are tracked through the instruction dependency graph and freed immediately after their last use, lowering GPU memory footprint and enabling larger batch execution.

- **Overlap of compute and memory**: Where possible, asynchronous data movement is overlapped with computation, ensuring GPU pipelines remain saturated.

6. Extensibility to Custom Hardware

Why Custom Hardware Matters for FHE

While GPUs provide the first practical acceleration path for FHE, they are not the endgame. FHE workloads are extremely arithmetic-intensive and structured, which makes them ideal candidates for domain-specific hardware. Several efforts are already underway to design FHE-oriented ASICs and FPGA implementations, targeting lower latency and higher throughput than general-purpose GPUs can offer.

However, these hardware efforts face a cold-start problem: without a mature software ecosystem and real workloads, hardware vendors cannot validate or commercialize their chips effectively. At the same time, enterprises cannot adopt FHE broadly if doing so requires waiting for specialized silicon.

Lattica solves this impasse by providing both: a production-grade FHE software platform today, and a portable abstraction layer for hardware tomorrow.

Portable Algorithmic Foundations

The portability of Lattica's system comes from how we formalized FHE execution as a lean set of tensor operations. Instead of exposing cryptographic primitives directly, we defined an interface of roughly 20 core functions over tensors that capture the entire space of operations needed for encrypted inference.

- These functions cover the essential homomorphic building blocks (modular arithmetic, rotations, rescaling, key switching components), but are expressed in a way that maps naturally to accelerators.

- By keeping the interface small and precise, we make it feasible for hardware vendors to implement full support without years of cryptographic engineering.

- Because ciphertexts are represented in our flexible tensor layout, with dedicated dimensions for polynomial degree and modulus chain, these operators can act directly on double-CRT data, enabling efficient homomorphic operations and batching without backend-specific rewrites.

This design strikes a balance: rich enough to express encrypted inference pipelines, lean enough to be portable across GPUs, TPUs, and custom FHE ASICs.

HEAL as the Hardware Integration Layer

To make this portability practical, we expose our system through the **Homomorphic Encryption Abstraction Layer (HEAL)**. HEAL provides the standard

contract between FHE algorithms and acceleration backends. For custom hardware vendors, HEAL means:

- They only need to implement a small set of primitive operators.
- They can integrate with Lattica's ecosystem without rebuilding client APIs, developer tools, or orchestration logic.
- Their hardware immediately becomes usable for real encrypted inference workloads.

This allows hardware companies to focus on raw performance at the silicon level while relying on Lattica to provide the runtime, APIs, and ecosystem adoption. To make integration practical, we provide a **complete suite** alongside the interface, including:

- A standalone runtime, so vendors can test their implementation in isolation.
- Unit tests that validate functional correctness.
- Example transcripts of computations, drawn from real encrypted inference workloads, to guide optimization and debugging.

Co-Design Opportunities

While the HEAL interface defines the core set of required functions, we also recognize that different hardware platforms have unique strengths. To enable innovation, the interface allows vendors to define additional custom operations that extend beyond the baseline.

For example, a chip might expose a fused polynomial-multiplication-and-rescale primitive, or specialized key-switch implementation. These vendor-specific extensions can be registered alongside the standard functions, giving hardware partners the flexibility to differentiate without breaking compatibility.

This approach ensures a common foundation across all backends, while giving each vendor room to highlight and exploit their hardware's unique advantages. In this way, Lattica remains complementary to hardware vendors, not competitive with them. We provide the execution layer that makes their innovations usable.

7. Query Client

While heavy homomorphic computations run on the Lattica platform, the query client handles all sensitive operations: key generation, encryption of inputs, and decryption of results.

Lightweight and Portable

The client is designed to run in constrained environments such as mobile devices, browsers, or enterprise endpoints:

- No custom HW required: Computation is optimized for CPUs, using multi-threading to take advantage of available cores.
- WebAssembly support: The client can be compiled to WebAssembly, enabling execution inside modern browsers with no installation overhead.

Open Source and Auditable

The query client is released as open source, allowing enterprises and regulators to audit its security. This transparency reinforces the zero-trust model: the client is the only component that ever handles plaintexts or secret keys, and its correctness can be independently verified.

Tensor-Based Design

Just like the server runtime, the query client represents data as tensors, built on top of Torch. This provides consistency across the stack: encryption, decryption, and homomorphic computation all share the same data abstraction. It also enables the client to apply analogous optimizations, such as thread-level parallelism and stride-based memory views, to avoid unnecessary copies and maximize efficiency on CPU-only environments.

8. Benchmarks & Results

Why Benchmarking Matters

For FHE to move from theory to production, performance must be demonstrated, not assumed. Academic libraries have long proven correctness, but they are orders of magnitude too slow for real-world AI workloads. Enterprises evaluating encrypted inference need to see concrete, reproducible benchmarks that compare:

- Latency: Can queries run in near real-time?
- Throughput: Can the system handle batch workloads at scale?
- Cost: Is the overhead low enough to be practical in the cloud?

Lattica has published head-to-head comparisons against academic baselines and cloud services. These results show that encrypted inference can be made both fast and cost-efficient, unlocking production use cases in healthcare, finance, and security.

Image Classification (CIFAR-10 Dataset)

We benchmarked encrypted inference on a CNN model against a widely cited baseline from the University of Delaware.

- Setup:
  - Dataset: CIFAR-10
  - Client: Python query client on a Linux PC, Intel i9-13900H.
  - Server: AWS p5.xlarge instance (NVIDIA H100 GPU).
  - Security: 228-bit parameters.

- Results:
  - University of Delaware baseline: 2,533 seconds per query (≈39 minutes), $5–10 cost.
  - Lattica: 31.5 seconds per query, $0.001 cost.

This represents a **~80× reduction in latency** and a **10,000× reduction in cost per query**, making encrypted image classification usable for interactive applications.

Try Our Live Demo

Image Classification - MNIST

Logistic Regression (Disease Prediction Dataset)

We also compared logistic regression inference, a widely used workload in healthcare and risk modeling, against a benchmark from the AWS SageMaker team.

- Setup:
  - Lattica dataset: disease prediction dataset (131 features, 41 classes).
  - AWS SageMaker dataset: Iris dataset (4 features, 3 classes).
  - Client: Python query client on a Linux PC, Intel i9-13900H.
  - Server: AWS p5.xlarge instance (NVIDIA H100 GPU).
  - Security: 228-bit parameters.

- Results:
  - AWS SageMaker baseline: 2,150 seconds for 5,000 queries (≈36 minutes).
  - Lattica: 6.08 seconds for 5,000 queries.

This demonstrates not only low per-query latency but also the ability to run large batch workloads efficiently.

Try Our Live Demo
Logistic Regression - Disease

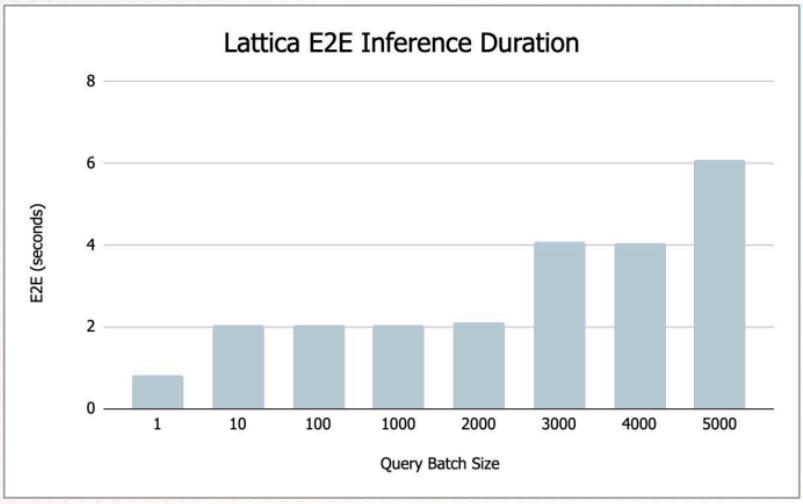
Nonlinear Scaling with Batching

A key advantage of Lattica's implementation is its ability to parallelize encrypted inference across batches of queries. Unlike the SageMaker benchmark, whose inference duration grows linearly with batch size due to limited parallelism, Lattica achieves nonlinear scaling by exploiting GPU parallelism and batch-aware homomorphic execution.

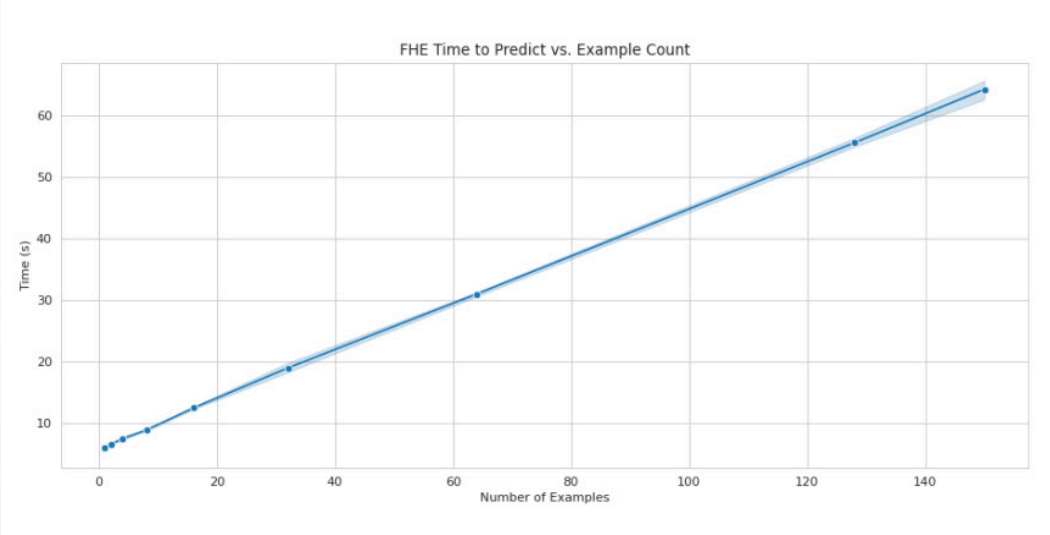
As shown in the figures below, increasing the query batch size from 1 to 5,000 leads to only a modest increase in total execution time. For example:

- A single query completes in under one second.
- A batch of 1,000 queries completes in ~2 seconds.
- Even 5,000 queries complete in just over 6 seconds.

This sublinear growth in runtime demonstrates that Lattica can serve both interactive, low-latency queries and large-scale batch jobs efficiently.



[SageMaker FHE Inference Duration]



Practical Takeaway

These benchmarks demonstrate that encrypted inference is both technically feasible and economically viable today. By combining FHE with GPU acceleration and batching, we achieve costs comparable to conventional cloud inference, even before specialized hardware is introduced.

- Queries can now be served in tens of seconds, not hours.
- Batch workloads complete in seconds, not half an hour.
- Costs are low enough to make real-world deployment practical.

For enterprises, this proves that privacy-preserving AI has moved beyond theory into deployment. For hardware vendors, it confirms that real workloads already exist today and can be accelerated even further tomorrow.

9. Security & Trust Model

Zero-Trust Architecture

Lattica's platform is designed around a zero-trust principle: the platform itself never sees plaintext data or decryption keys.

- Client: Responsible for generating and holding cryptographic keys, encrypting inputs, and decrypting results. Keys never leave the client environment.

- Lattica Platform: Executes encrypted inference entirely on ciphertexts. The platform processes only encrypted data and evaluation keys, which are sufficient for computation but do not allow decryption.

This separation ensures that even though computations are executed on the Lattica platform, sensitive data remains cryptographically protected at all times.

Key Management

Key management is performed fully on the client side:

- Secret keys are generated and stored locally by the end-user.
- Only evaluation keys, which are special-purpose keys required to enable homomorphic operations, are uploaded to the Lattica platform.
- These evaluation keys cannot be used to recover plaintext, ensuring that decryption capability always remains solely with the end-user.

This guarantees that the customer retains full cryptographic control, even while leveraging the Lattica platform for execution.

Cryptographic Foundations

Lattica's platform is based on state-of-the-art lattice-based encryption schemes, specifically:

- CKKS, which supports approximate arithmetic and is well-suited for deep learning and statistical models.
- BGV, which supports exact modular arithmetic and is useful for workloads requiring discrete, precise computation.

Both schemes are widely studied in the cryptography community and are regarded as the most practical FHE schemes for real-world deployment.

Threat Model

We explicitly design against the following adversarial threats:

- Untrusted platform operators: Even though inference runs on the Lattica platform, ciphertexts cannot be decrypted without client-held keys.

- Hardware-level vulnerabilities: Lattica's model does not rely on trusted execution environments (TEEs) or other hardware-based secrecy, and is therefore robust against side-channel or firmware-level exploits.

- Malicious insiders: No Lattica operator or administrator has access to decryption keys or plaintexts.

We assume that client endpoints remain secure and uncompromised; key leakage at the client would undermine the security guarantees of any FHE system.

10. Developer Experience

Familiar ML Abstractions

One of the major barriers to adopting FHE has been that existing libraries expose cryptographic primitives rather than developer-friendly APIs. Most ML engineers are not cryptographers; they work with tensors, models, and inference pipelines.

Lattica bridges this gap by exposing FHE through abstractions that mirror familiar ML workflows. Developers continue to work with models in a style similar to PyTorch, while the complexity of homomorphic execution remains hidden.

This lowers the barrier to adoption dramatically: teams can build encrypted inference pipelines without retraining their engineers in cryptography.

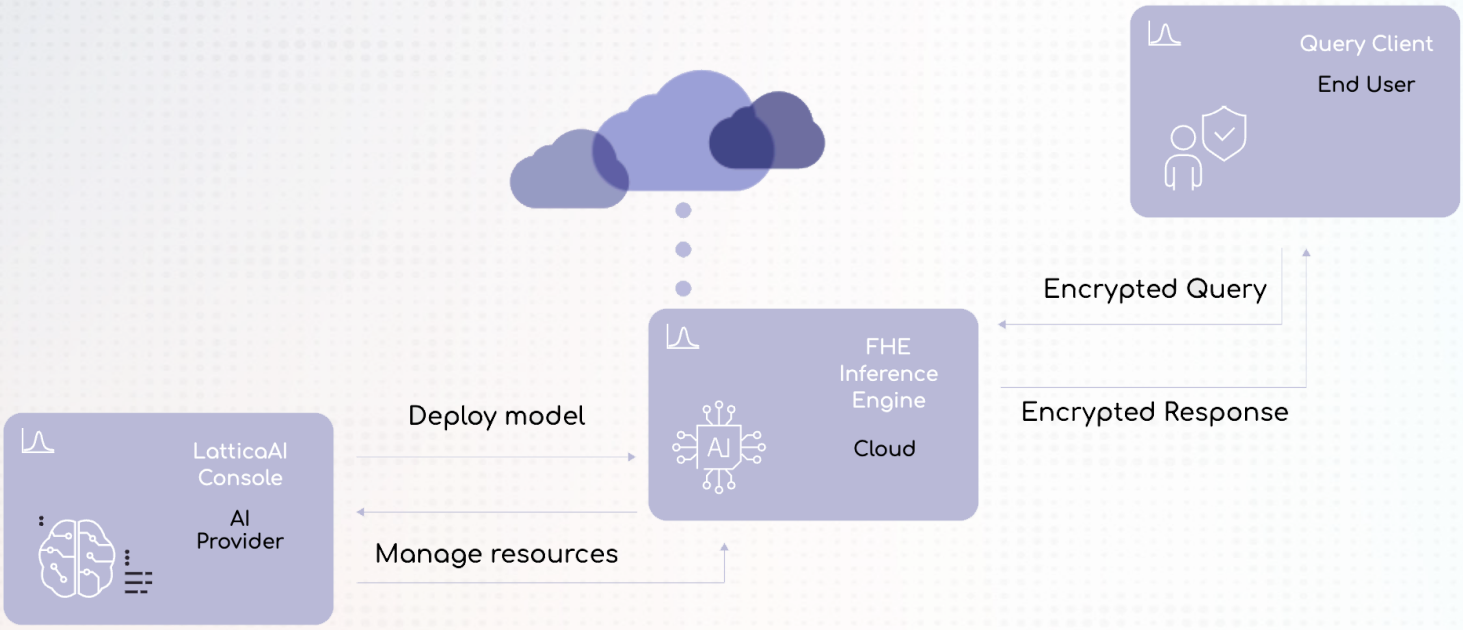
API and SDK

Lattica provides an API-first experience:

- Model Deployment: AI providers can upload trained models through the Lattica console or SDK, where they are adapted for encrypted inference.

- Encrypted Inference: End-users (or enterprise clients) can query these models with encrypted inputs, receiving encrypted outputs.

SDKs are available in Python and TypeScript, making it easy to integrate into research prototypes and production systems alike.

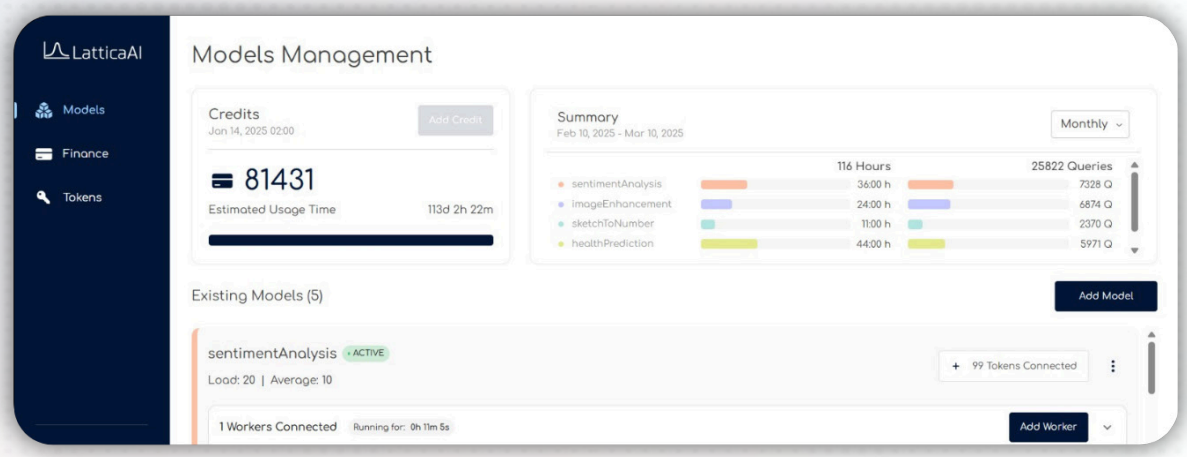


Console and Management Tools

The **Lattica Console** provides operational visibility and governance:

- **Model lifecycle**: AI providers can upload, manage, and monitor models.
- **Access control**: Usage can be restricted through policies and tracked for auditing.
- **Performance metrics**: Query latency, throughput, and cost are visible in real time.
- **Compute management**: Enterprises can allocate, scale, and monitor compute resources, ensuring optimal performance and cost efficiency.

This ensures that encrypted inference can be deployed and operated with the same rigor as standard cloud ML services.



Example Workflows

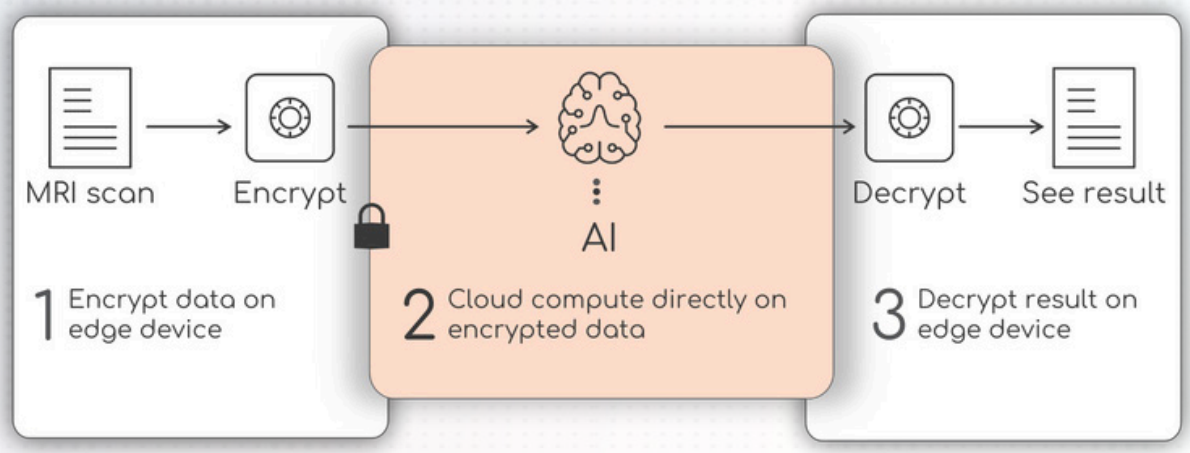
AI Provider (Model Owner)

1. Train a model in their preferred ML framework.
2. Upload the model to the Lattica platform via console or SDK.
3. Lattica adapts the model into an encrypted inference flow, using CUDA/Torch-optimized FHE kernels.
4. The provider can manage usage and monitor performance through the console.

Enterprise Client (Query User)

1. Encrypts input data locally using the Lattica SDK (e.g., a medical scan, financial record).
2. Sends the encrypted input to the Lattica platform.
3. The platform executes the encrypted inference and returns an encrypted result.
4. The client decrypts the result locally to obtain the plaintext prediction.

In both cases, plaintext data and secret keys never leave the client environment, ensuring end-to-end privacy.



Benefits for Developers

This developer experience offers:

- Minimal friction: Model owners and data owners both integrate with simple APIs.
- Operational control: Providers manage models, clients retain key ownership.
- Security by default: Neither side needs to handle cryptographic details; the platform enforces them automatically.

By separating roles but unifying workflows under familiar ML abstractions, Lattica makes encrypted inference practical for both AI providers and data owners.|

11. Conclusion

Fully Homomorphic Encryption has long promised a future where sensitive data can be processed without ever being exposed. Until now, that promise has been limited by prohibitive performance costs, lack of usable developer tools, and no clear path to hardware integration.

Lattica changes this by:

- Re-engineering FHE primitives for accelerators, with CUDA/Torch implementations that achieve orders-of-magnitude performance improvements.
- Providing a zero-trust platform where sensitive data never leaves the encrypted domain.
- Exposing encrypted inference through familiar ML abstractions, enabling adoption by developers without cryptographic expertise.
- Delivering a portable interface (HEAL) that allows new hardware vendors to join the ecosystem quickly and confidently.

Encrypted inference is no longer theoretical, it is deployable, performant, and economically viable. By bridging the gap between cryptography and acceleration, Lattica is making FHE practical for the first time, and building the foundation for a world where organizations can apply AI to their most valuable data without ever exposing it.

www.lattica.ai