Andrei Stoian

May 2024

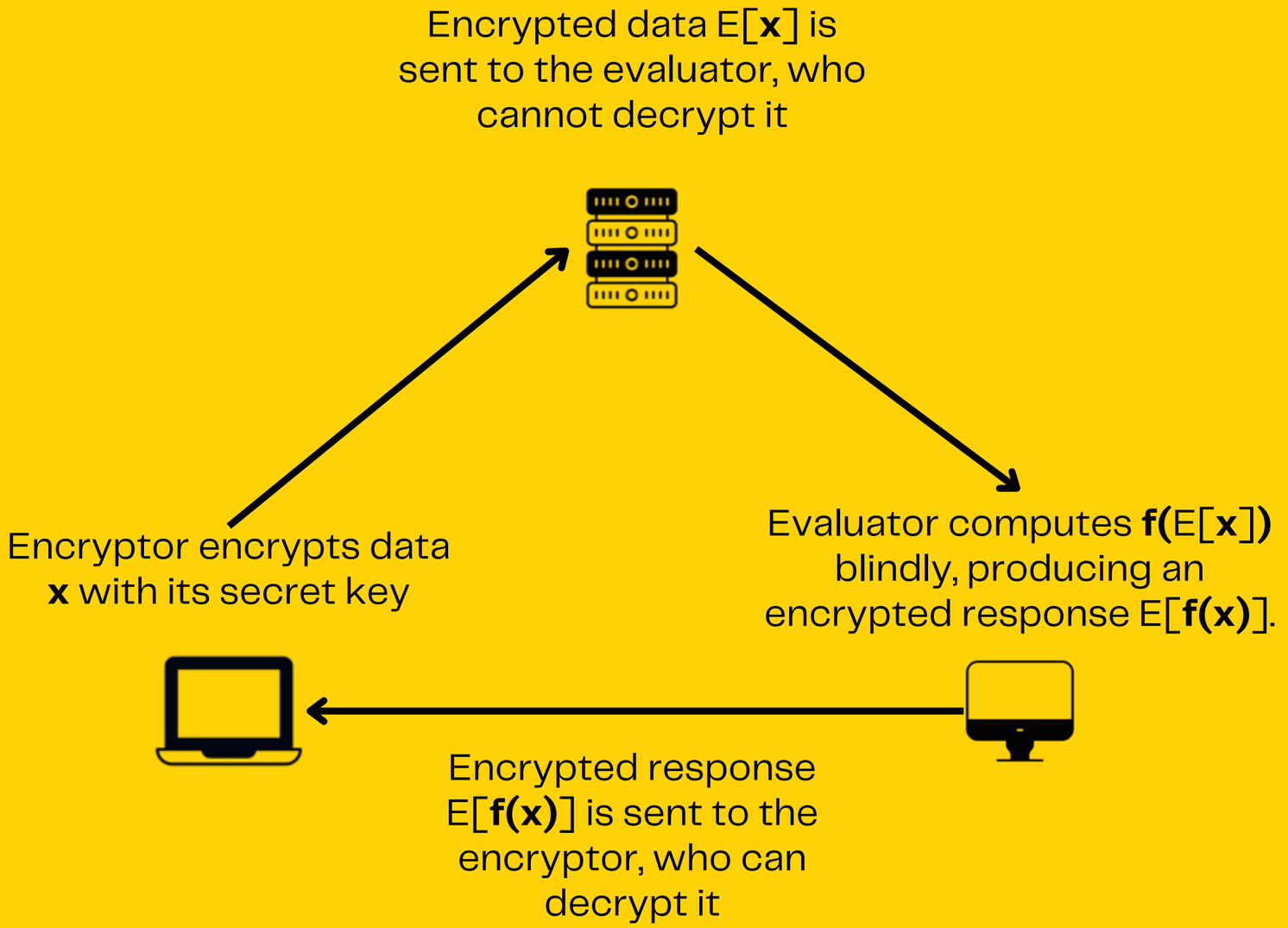
Concrete ML

Privacy Preserving Machine Learning with Fully Homomorphic Encryption

ZAMA

Zama is a **cryptography company** providing **open source homomorphic encryption solutions** for **blockchain and AI.**

Fully Homomorphic Encryption (FHE) enables processing data blindly



FHE enables encrypted data processing

$$E[x] + E[y] = E[x + y]$$

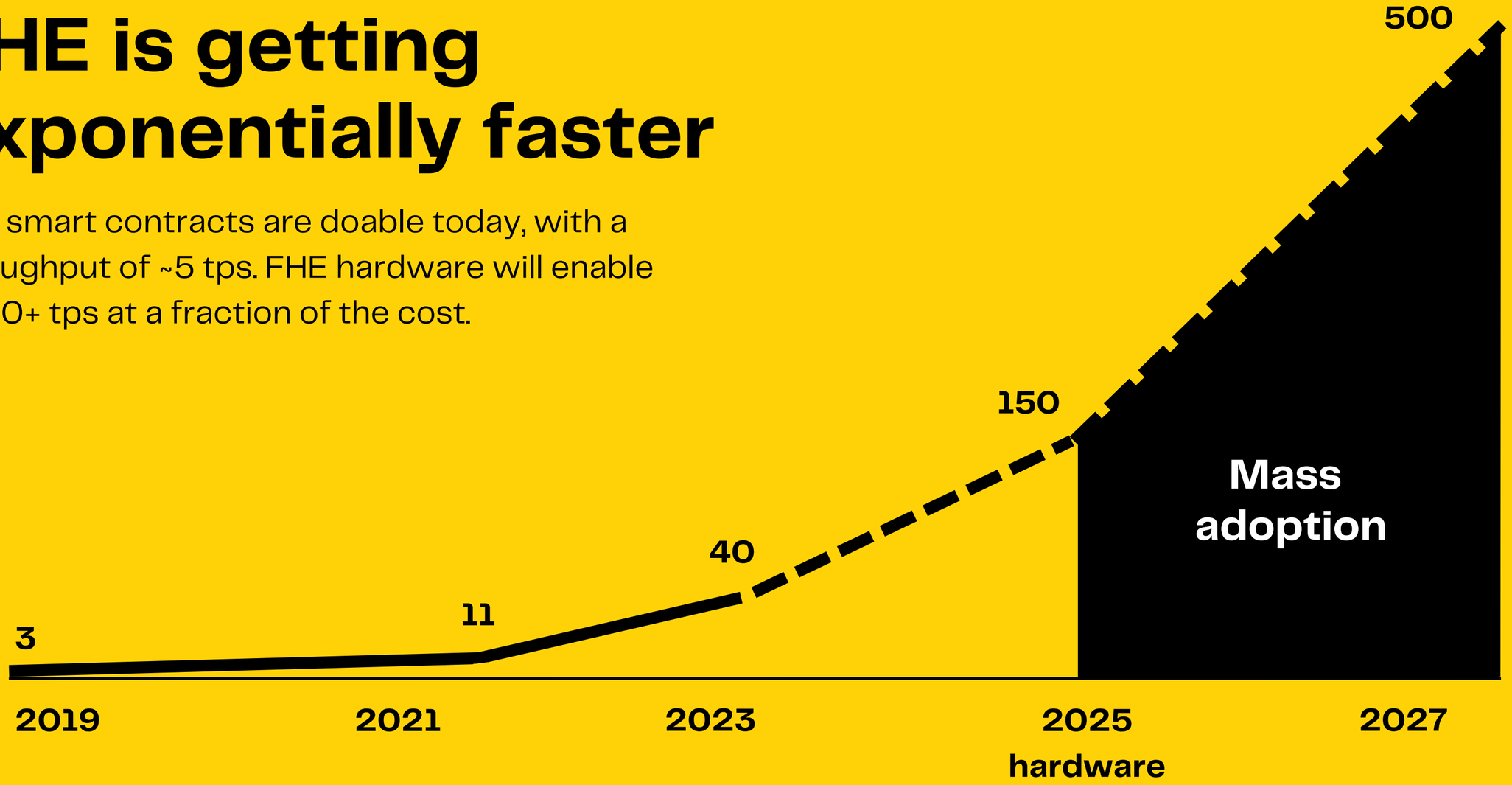
$$E[x] < E[y] = E[x < y]$$

More generally:

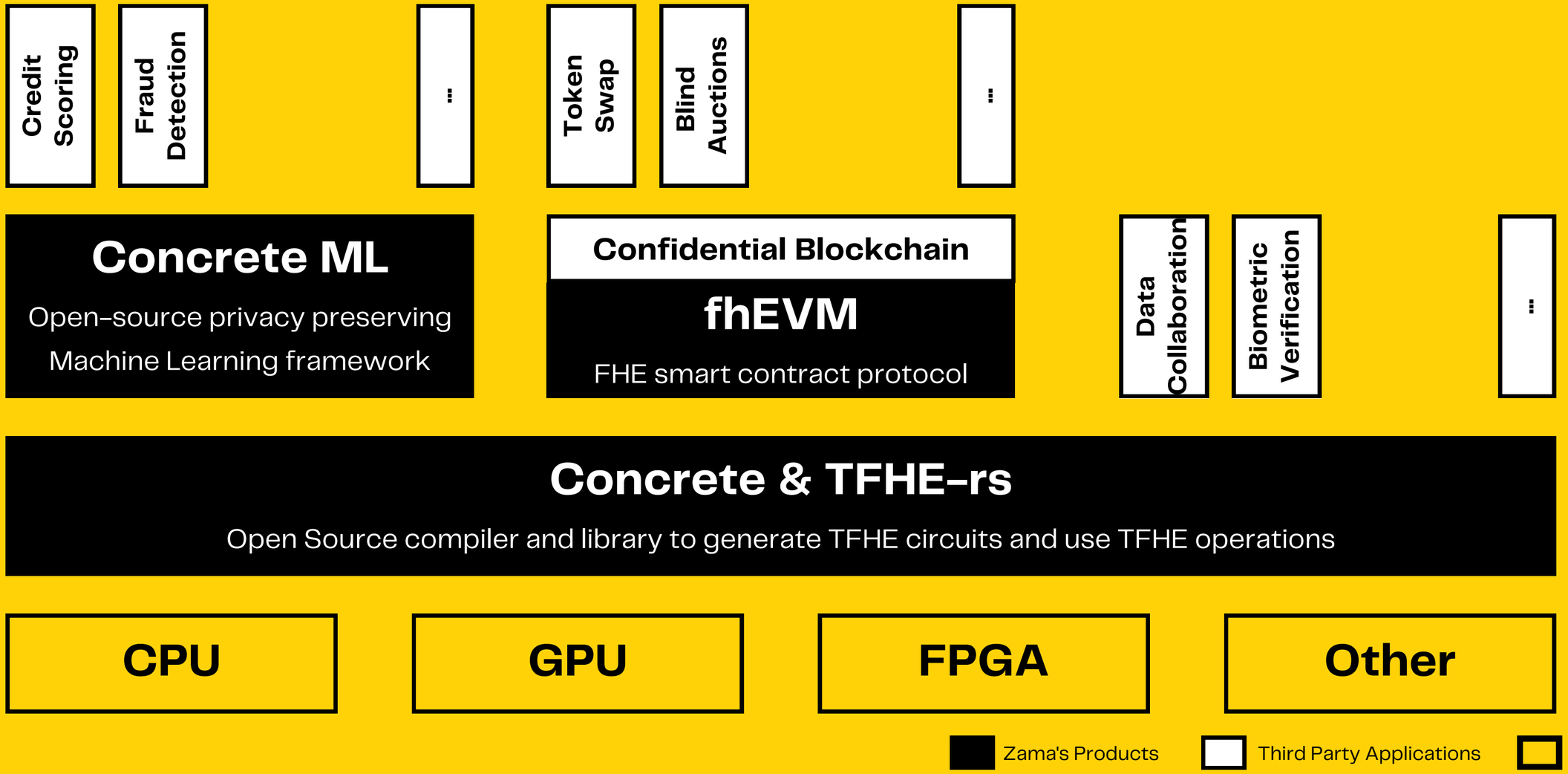
$$f(E[x], ..., E[y]) = E[f(x, ..., y)]$$

FHE is getting exponentially faster

FHE smart contracts are doable today, with a throughput of ~5 tps. FHE hardware will enable 1,000+ tps at a fraction of the cost.



Our products enable secrecy-preserving, regulatory-compliant applications



Concrete ML

Concrete ML enables data scientists to work with sensitive data while protecting their models

Concrete ML

When should you consider confidential ML?

- Working with private or confidential data
- Working with regulated data subject to data residency
- You care about data privacy
- The model has sensitive IP

Concrete ML

What can confidential ML do for you?

- Train a model without seeing the confidential training data
- Run inference without seeing the confidential data or the inference results
- Protect the intellectual property of your model

How does Concrete ML achieve confidentiality?

- Data and results are always encrypted with post-quantum encryption FHE allows the processing of encrypted data without decrypting it
- No need to trust a hardware manufacturer
- No need to manage multiple, independent trusted entities

Concrete ML

Use-cases

Private Inference

Secure Collaboration

Private Training

Private LLMs

Concrete ML: Powerful features are available out of the box

Supports many types of models
Linear models, XGBoost, trees, neural networks...

Unlimited number of layers
Unlimited depth without FHE noise accumulation

Support for any activation function
Use any non-linear activation without approximation

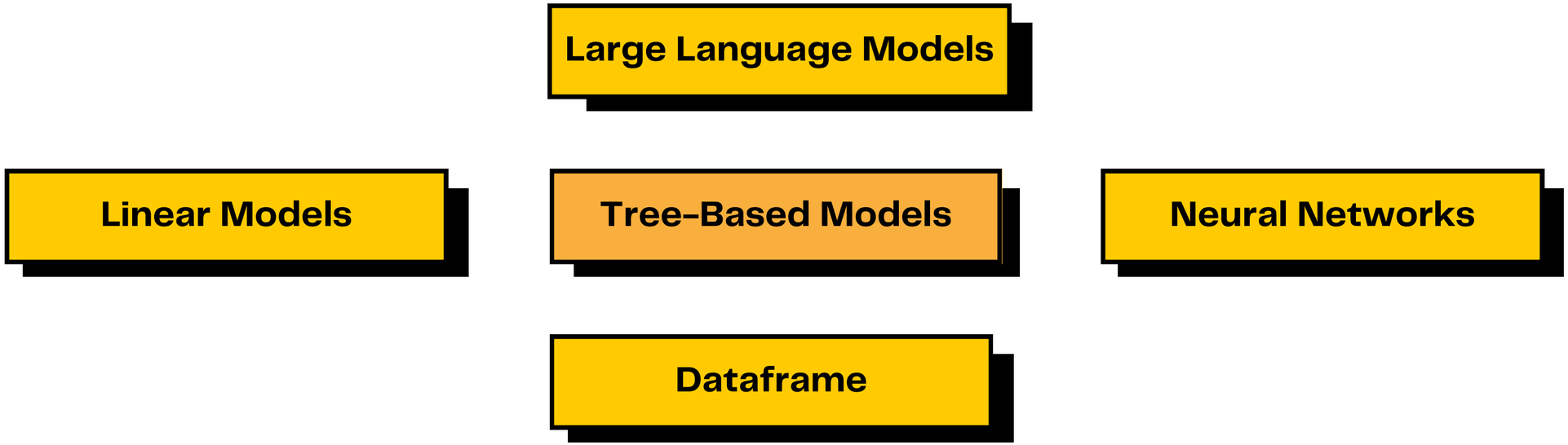
Automatic selection of FHE parameters
Model-derived FHE quantization and parametrization

Exact results, just like in plaintext
FHE output is exactly the same as plaintext one

Confidential (& collaborative) Training
Train in FHE or combine with Federated Learning

Concrete ML

- An open-source library for data scientists, for PPML
- Built for data scientists on top of scikit-learn and torch
- No need to know cryptography



Easy to use APIs

```python
from concrete.ml.sklearn import LogisticRegression

model = LogisticRegression(n_bits=12)
model.fit(X_train, y_train)
model.predict(X_test)
model.compile(X_train)
model.predict(X_test, fhe="simulate")
model.predict(X_test, fhe="execute")
```

```python
from concrete.ml.sklearn import XGBClassifier

model = XGBClassifier(n_bits=8)
model.fit(X_train, y_train)
model.predict(X_test)
model.compile(X_train)
model.predict(X_test, fhe="simulate")
model.predict(X_test, fhe="execute")
```

No need to know cryptography!

Easy to use APIs

Custom Neural Networks

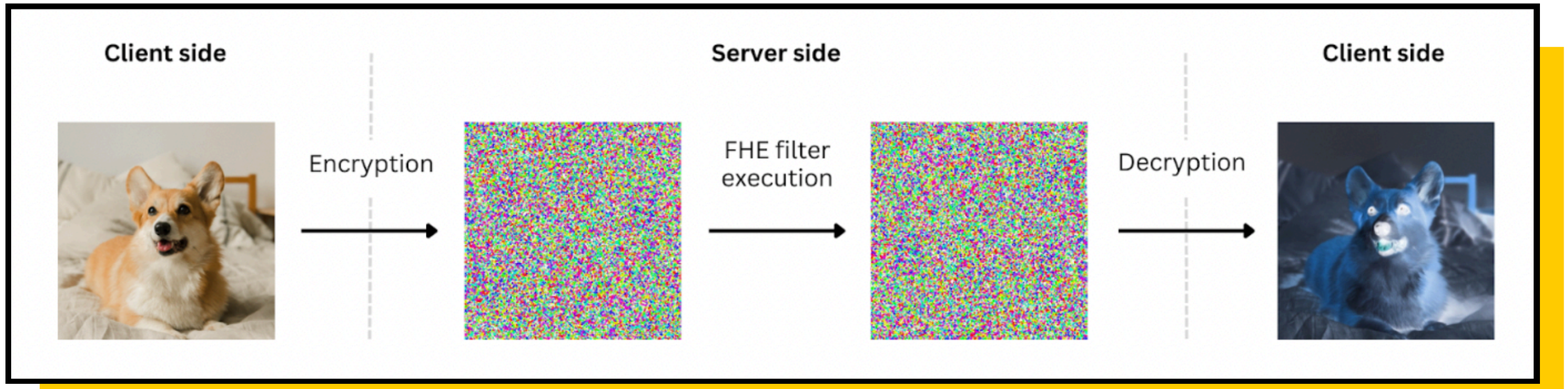
Tensorflow
Torch
ONNX

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from concrete.ml.torch.compile import compile_torch_model

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 30)
        self.fc2 = nn.Linear(30, 30)
        self.fc3 = nn.Linear(30, 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = SimpleNet()
input_data = torch.randn(100, 784)

quantized_fhe_module = compile_torch_model(model, input_data, n_bits=8)
```

Easy to use APIs

Image Filtering

A Hugging Face space to apply filters over images homomorphically using the developer-friendly tools in Concrete-Python and Concrete-ML.



huggingface.co/spaces/zama-fhe/encrypted_image_filtering

Easy to use APIs

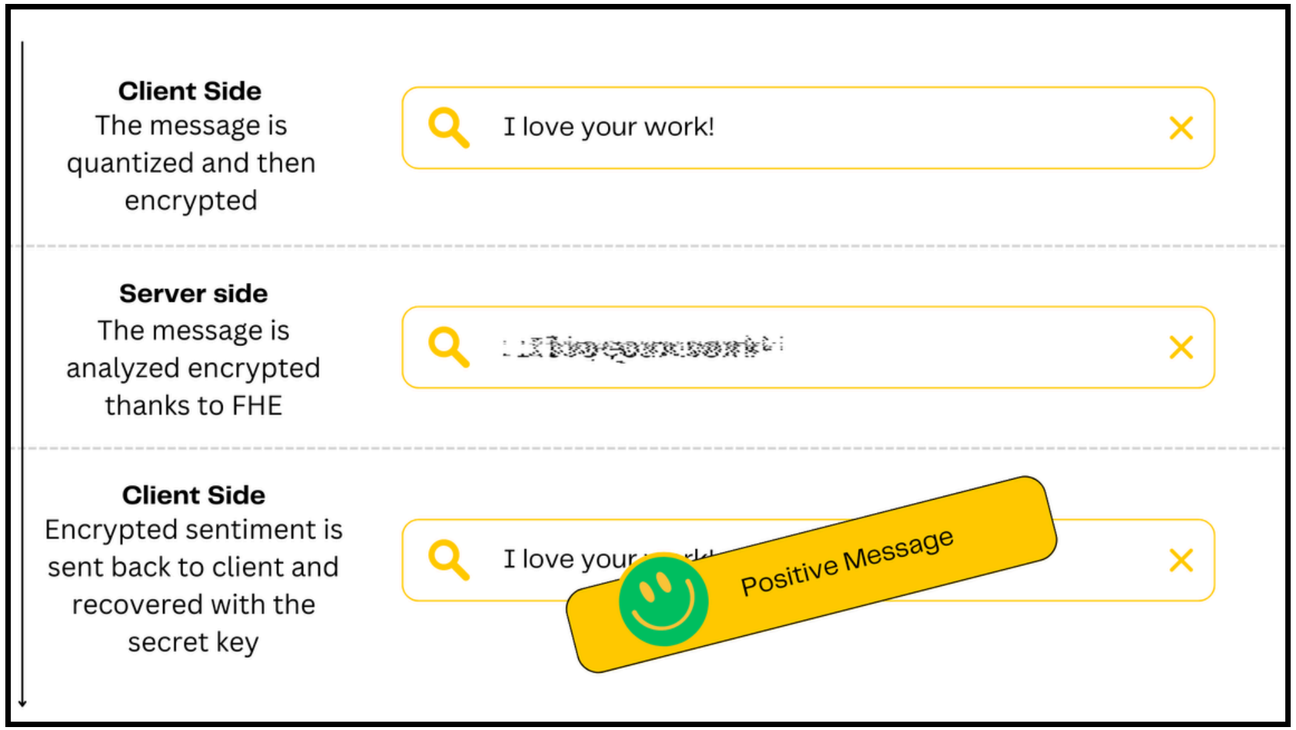
Image Filtering

```python
def forward(self, x):
    """Forward pass with a single convolution using a 2D kernel."""
    # Reshape the input to (1, 3, Height, Width)
    x = x.transpose(2, 0).unsqueeze(axis=0)

    # Apply the convolution
    x = nn.functional.conv2d(
        x,
        self.kernel,
        stride=stride,
        groups=self.groups
    )

    # Reshape the output to (Height, Width, 3)
    x = x.transpose(1, 3).reshape((
        x.shape[2],
        x.shape[3],
        self.n_out_channel
    ))

    return x
```

huggingface.co/spaces/zama-fhe/encrypted_image_filtering

Easy to use APIs

Sentiment Analysis



huggingface.co/spaces/zama-fhe/encrypted_sentiment_analysis

Easy to use APIs

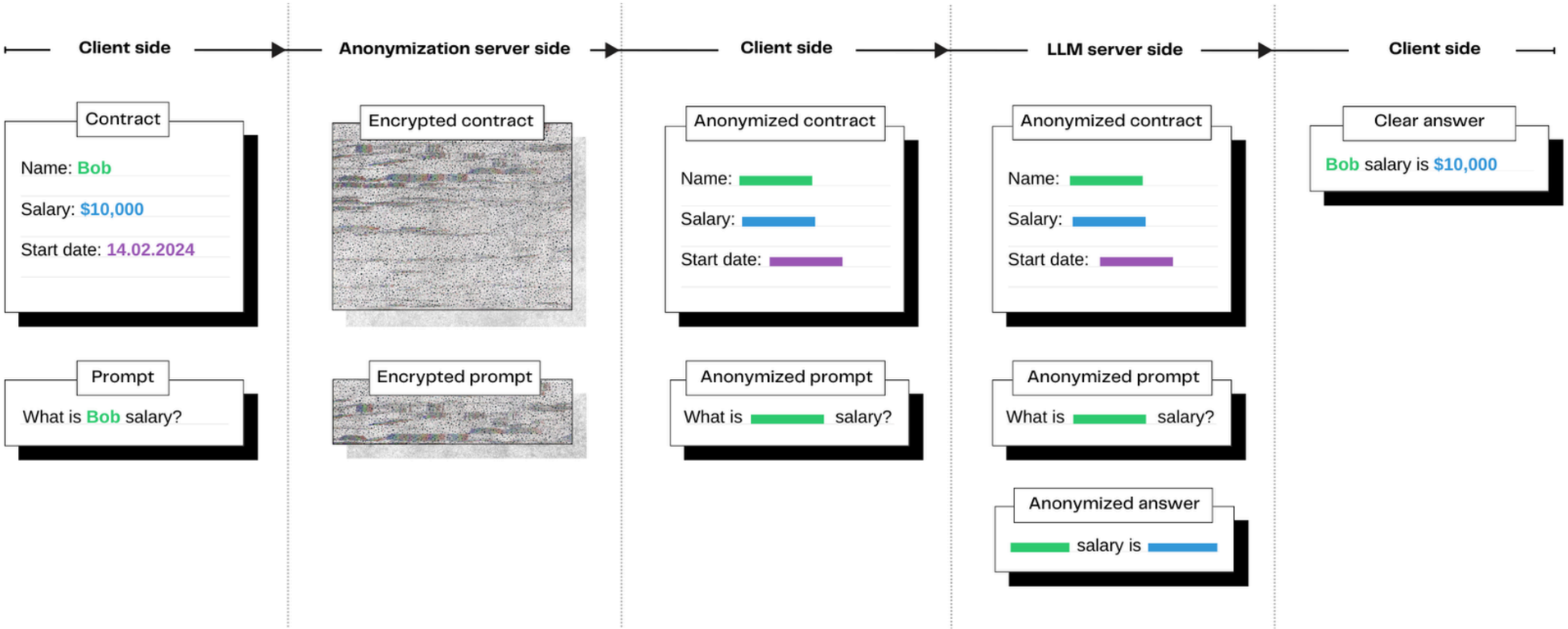
Sentiment Analysis

```python
from concrete.ml.sklearn import XGBClassifier
from concrete.ml.deployment import FHEModelDev

# Let's build our model
model = XGBClassifier()

# A gridsearch to find the best parameters
parameters = {
    "n_bits": [2, 3],
    "max_depth": [1],
    "n_estimators": [10, 30, 50],
}

# Train the model on the parameter grid
grid_search = GridSearchCV(model, parameters, cv=3, scoring="accuracy")
grid_search.fit(X_train, y_train)

# Compile the model to work on encrypted data
best_model.compile(X_train)

# Let's save the model to be pushed to a server later
fhe_api = FHEModelDev(DEPLOYMENT_DIR / "sentiment_fhe_model", best_model)
fhe_api.save(via_mlir=True)
```

huggingface.co/spaces/zama-fhe/encrypted_sentiment_analysis

Easy to use APIs

Anonymized ChatGPT queries



https://huggingface.co/spaces/zama-fhe/encrypted-anonymization

Easy to use APIs

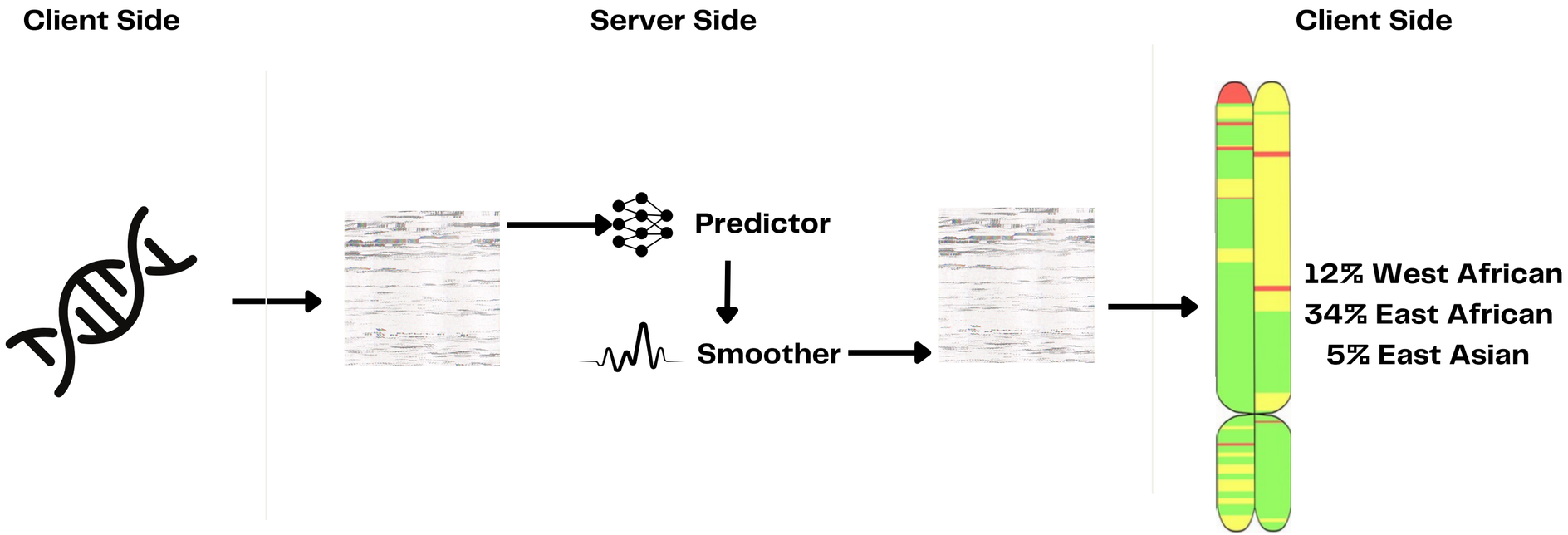
Anonymized ChatGPT queries

```python
def encrypt_query(self, text: str):
        # Pattern to identify words and non-words (including punctuation, spaces, etc.)
        tokens = re.findall(r"(\b[\w\.\/\-@]+\b|[\s,.!?;:'\"-]+)", text)
        encrypted_tokens = []

        for token in tokens:
            # Directly append non-word tokens or whitespace to processed_tokens
            if bool(re.match(r"^\s+$", token)):
                continue

            # Extract features for each word
            emb_x = get_batch_text_representation([token], self.embeddings_model, self.tokenizer)

            # Encrypt each word
            encrypted_x = self.client.quantize_encrypt_serialize(emb_x)

            encrypted_tokens.append(encrypted_x)

        write_pickle(KEYS_DIR / f"encrypted_quantized_query", encrypted_tokens)
```

https://huggingface.co/spaces/zama-fhe/encrypted-anonymization

Easy to use APIs

Private DNA Ancestry



Easy to use APIs

Private DNA Ancestry

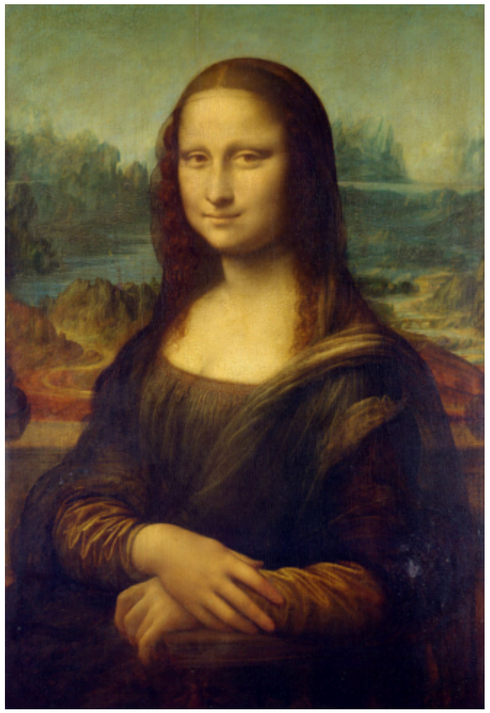
Model definition

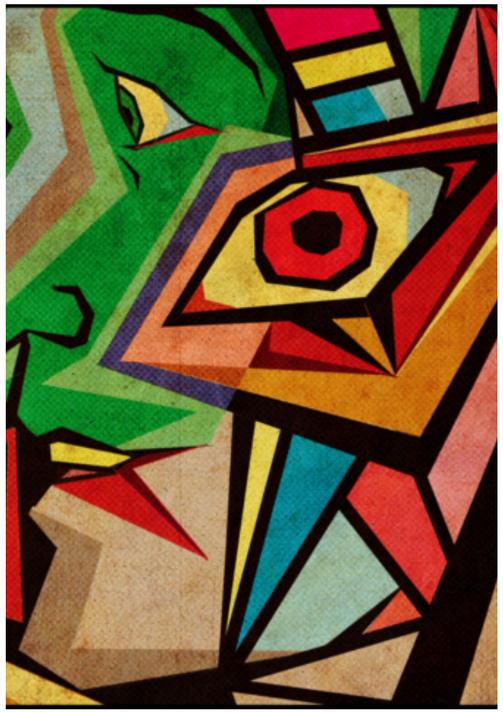
Model training

```python
from concrete.ml.sklearn import XGBClassifier, LogisticRegression

base_models = [LogisticRegression(n_bits=8, penalty="l2", C=3., max_iter=1000) for _ in range(n_windows)]
smoother = XGBClassifier(
    n_bits=4, n_estimators=100, max_depth=4,
    learning_rate=0.1, reg_lambda=1, reg_alpha=0,
    n_jobs=N_JOBS, random_state=SEED,
    use_label_encoder=False, objective='multi:softprob',
)

def train_base_models(models, X_t, y_t):
    # Build sliding windows
    M_ = meta["M"] + 2 * context
    idx = np.arange(0, meta["C"], meta["M"])[:-2]
    X_b = np.lib.stride_tricks.sliding_window_view(X_t, M_, axis=1)[:, idx, :]

    # Configure training arguments for each window
    train_args = tuple(zip(models[:-1], np.swapaxes(X_b, 0, 1), np.swapaxes(y_t, 0, 1)[:-1]))
    rem = meta["C"] - meta["M"] * n_windows
    train_args += ((models[-1], X_t[:, X_t.shape[1] - (M_ + rem):], y_t[:, -1]),)

    # Train each base model
    log_iter = tqdm(train_args, total=n_windows, position=0, leave=True)
    return [b[0].fit(b[1], b[2]) for b in log_iter]

def train_smoother(smoother, X_t, y_t):
    X_slide, y_slide = slide_window(X_t, SMOOTH_SIZE, y_t)
    return smoother.fit(X_slide, y_slide)
```

Easy to use APIs

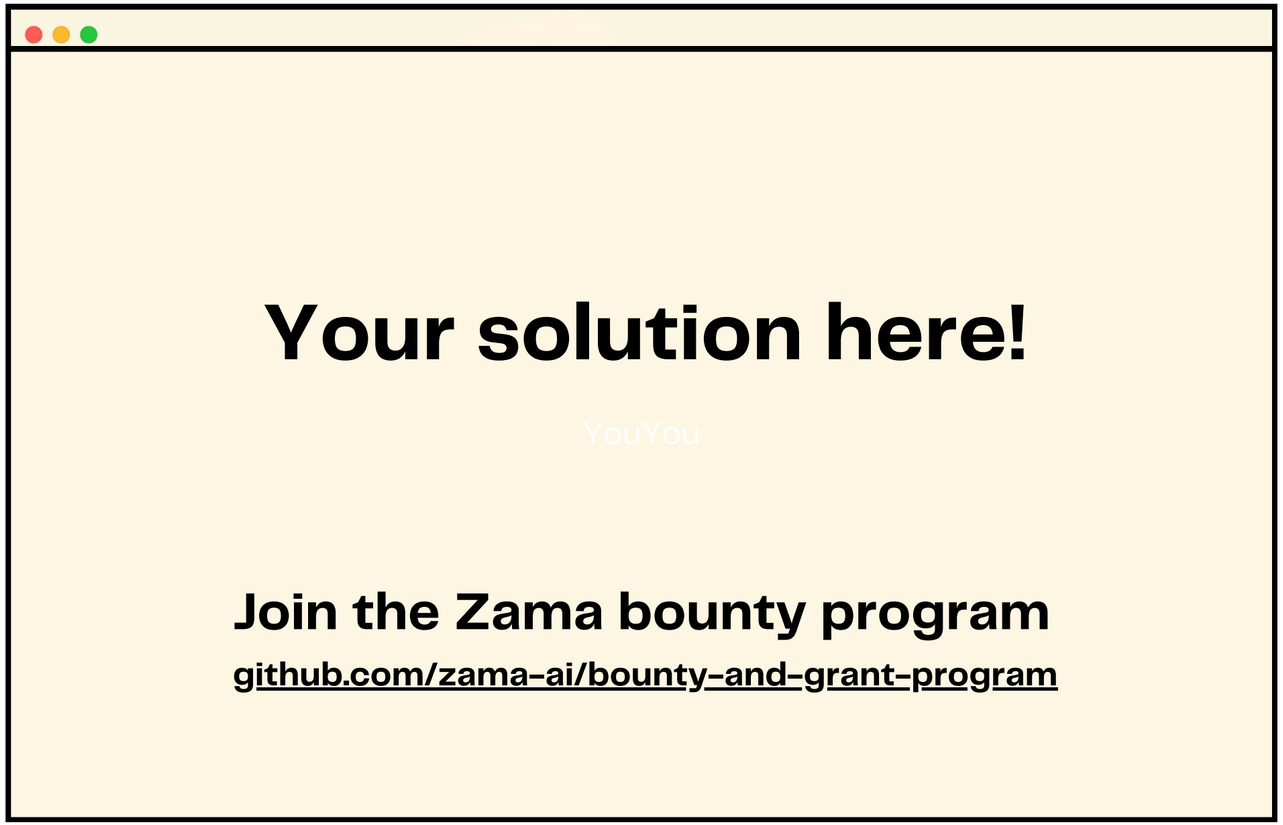
Style transfer





Easy to use APIs

Style transfer



Contact and Links

zama.ai

Github

Community links

ZAMA