

Ciphertext and Encoding Type Detection

Microservice

Overview

Identifying the encoding or encryption behind a mysterious string is a common challenge in CTF platforms. The goal is to build a “magic” detector (akin to CyberChef’s **Magic** function) that can **classify an input string’s format** – whether it’s a simple encoding (Base64, hex, etc.), compressed data, or ciphertext from a specific algorithm (AES, RSA, XOR, classical ciphers, etc.). Modern approaches fall into three broad categories: (1) **heuristic-based detectors** using pattern recognition and statistics, (2) **machine learning models** trained to recognize cipher/encoding patterns, and (3) **hybrid systems** combining both. Below, we detail state-of-the-art methods in each category, along with notes on accuracy, deployment feasibility, and how they can integrate into a microservice. We also suggest datasets for training/evaluation and an output format for the service’s responses.

1. Heuristic-Based Detection

Heuristic methods use predefined patterns and simple analyses to guess the data format. These are **fast, interpretable, and work well for known encoding schemes**:

- **Pattern Matching (Regex & Signatures):** Many encodings and some ciphertexts have telltale patterns. For example, **Base64** strings consist of `[A-Za-z0-9+/]` characters and often end with `=` padding. CyberChef’s “Magic” uses regular expressions for such patterns (e.g. a regex for standard Base64: `^(?:[A-Z\d+/]{4})+(?:[A-Z\d+/]{2}==|[A-Z\d+/]{3}=)?$` ¹). Similarly, hex-encoded data matches `^[0-9A-Fa-f]+$` (often an even length). Gzip-compressed data can be recognized by specific **magic bytes** (the first two bytes `0x1F 0x8B` in the GZIP format) ². By checking for these patterns, a detector can immediately flag known formats with high confidence. Tools like **PyWhat** utilize hundreds of such patterns to identify hashes, encodings, UUIDs, etc., by matching against a library of regexes and known byte signatures ³. Integrating these pattern-based checks provides quick wins (e.g. “this string matches Base32 alphabet” or “this bytes sequence starts with common PNG header bytes”).
- **Entropy and Randomness Tests:** Encrypted or compressed data tends to have high Shannon entropy (appear random), whereas plaintext or simple encodings often have lower entropy. A heuristic detector can calculate the input’s entropy as a numeric score ⁴. **High entropy** suggests the data could be ciphertext (or compressed) – i.e., not easily distinguishable from random noise. Conversely, if entropy drops significantly after a decoding step, it indicates the result might be structured (which is good, as it might be the plaintext or an intermediate readable format). This heuristic helps decide if an output “looks like” meaningful data or still like ciphertext. For instance, CyberChef Magic ranks decoding branches that produce lower-entropy output higher, under the assumption that **structured data (like text) has lower entropy than truly random bytes** ⁵.

- **Language and Frequency Analysis:** If the data (or some decoded layer) is text, we can check how closely it resembles natural language. A classic technique is **chi-squared analysis on character frequencies** ⁶. English text has a very distinct letter frequency distribution (e.g., 'E' is most common, 'Q' is rare, etc.). By comparing the byte/letter frequency of a decoded blob to that of English (or other languages), one can gauge if it's likely plaintext. CyberChef's Magic uses a precomputed frequency table from Wikipedia text and applies Pearson's chi-squared test to measure goodness-of-fit to English ⁷ ⁸. A low chi-squared score means the text "looks like English," indicating a successful decode. This method is effective for detecting outputs of classical ciphers or encodings – for example, a Caesar-shifted message will still roughly follow English letter frequencies and score well. Tools like **Ciphey** combine chi-square tests with dictionary checks (seeing if many real words appear) to decide if a decoded string is valid plaintext ⁹ ¹⁰. Using such checks in the detector allows it to not only identify encodings but also to **verify when a decryption/decoding was successful** (the text turns to readable English or another language).

- **Heuristic Identifiers for Specific Formats:** Some data formats announce themselves. For example, an RSA public key in PEM format literally contains the header `-----BEGIN RSA PUBLIC KEY-----`. A JWT token can be spotted by its structure of base64 strings separated by dots. Even without such clear markers, context clues like allowable character set (e.g., base58 vs base64 alphabets) or length (hex-encoded MD5 hashes are 32 hex characters) can give them away. Simple rules can detect these: e.g., **Hash Identification** by length – a 64-character hex string is likely a SHA-256 hash, 40-character hex could be SHA-1, etc. (tools like HashID or PyWhat use such rules). Another example: if a binary blob starts with common file signatures (PK\x03\x04 for ZIP, or PDF's `%PDF-`), a heuristic could label the data as "Zip archive" or "PDF document" accordingly. Integrating a **file-signature library (libmagic)** in the pipeline can help recognize if decoded bytes correspond to a known file type.

Advantages: Heuristics are **lightweight and fast**, making them ideal for real-time use in a microservice. They are human-understandable (each rule is transparent) and can assign **confidence scores** in straightforward ways (e.g., a full regex match = high confidence, a partial or entropy-based guess = moderate confidence). Indeed, CyberChef's Magic operation is largely heuristic: it systematically tries regex detections and decodes, then measures entropy/chi-square to decide if the result is likely correct ¹¹ ¹². This approach works very well for **common encodings and many classical ciphers** that have known patterns. We recommend implementing a similar heuristic library in the microservice as the first detection layer (e.g., regex checks for Base64/hex/Base32, scanning for known file magic bytes after decoding, computing entropy and language scores, etc.). These can immediately label obvious cases with near-100% confidence (for instance, detecting a Base64 string with proper padding and character set ¹). Heuristics will also provide interpretable reasons (useful for logging or user output, e.g. "Detected Base64 due to character set and padding").

Limitations: Pure heuristics might struggle with data that doesn't match known patterns or is intentionally obfuscated to evade simple detection. They also need to be maintained as new encodings/formats emerge. This is where a learning-based approach can complement the system.

2. Machine Learning Models for Format Classification

Machine learning offers a way to **automatically learn the features of different ciphertexts or encodings**, potentially catching things that are hard to manually encode as rules. In recent research, ML models (from classic algorithms to deep neural networks) have been applied to classify cipher types with notable success:

- **Feature-Based ML (SVM/Ensemble):** Earlier approaches treated ciphertext identification as a standard classification task by extracting statistical features and feeding them to a classifier. For example, one project extracted 11 features (like number of unique characters, character frequency distribution, index of coincidence, etc.) from text, then used an ensemble of SVMs to decide the cipher type ¹³. Each cipher had distinct feature patterns; e.g., a simple substitution cipher preserves letter frequency (just permuted), whereas a Vigenère cipher flattens frequency distribution, and a one-time pad or modern cipher produces near-uniform frequency. By choosing discriminative features, such systems achieved decent accuracy (often in the 70-80% range for classical cipher families) with minimal runtime cost. These classical ML models are quite lightweight – an SVM model with a few dozen features can run in microseconds – so they are suitable for local deployment. The downside is they require domain expertise to select features and may not generalize to formats not originally considered.
- **Deep Learning (CNNs and RNNs):** Modern deep learning can **learn features directly from the raw data (text or bytes)**. Convolutional Neural Networks (CNNs) can be applied to ciphertext by treating the text as a one-dimensional image or byte sequence. For instance, one could feed in the ciphertext's byte values (or an image like a histogram) into a CNN to learn patterns. Some studies even converted text to visual representations (like histograms or “punched tape” images of bits) and then used image-classifying CNNs ¹⁴. Recurrent Neural Networks like LSTMs or GRUs, on the other hand, treat the data as a sequence, potentially capturing patterns like repeating cycles (useful for detecting polyalphabetic ciphers with certain key lengths) or specific byte dependencies. A 2022 experiment built a deep bidirectional LSTM-GRU network for automated ciphertext classification, showing that RNNs can achieve strong results across multiple cipher types ¹⁵. The advantage of RNNs/CNNs is that they can detect subtle statistical quirks of different algorithms (for example, the presence of small-scale repetitions, or the distribution of bigrams/trigrams) without explicitly coded rules.
- **Transformers and BERT:** The latest and often most accurate models use Transformer-based architectures (the kind behind BERT, GPT, etc.). Transformers are powerful at modeling sequences and have been adapted for ciphertext detection by treating the ciphered text as a “language” of its own. Researchers have fine-tuned transformer models (like BERT) to classify cipher types, effectively learning the “language of ciphertexts.” In a 2024 study, Sikdar et al. compared CNNs, Transformers, and BERT models for cipher identification and found that **a BERT-based model outperformed the others in accuracy** ¹⁶. Essentially, BERT's rich sequence representation could distinguish cipher texts more reliably. Another work on 55 classical ciphers reported ~72% accuracy using a Transformer on 100-character samples ¹⁷. Although that was slightly below a carefully crafted feature-based method, they noted that combining the two closed the gap (we'll discuss hybrid ensemble next) ¹⁸. The takeaway is that **state-of-the-art ML models (especially transformers)** can achieve high accuracy in labeling an unknown ciphertext/encoding. These models output probabilities for each class, which naturally serve as **confidence scores** for the prediction (e.g., “model is 95% sure this is Base64 vs 5% hex”). Confidence calibration can be further improved by

techniques like temperature scaling if needed, but generally the softmax probabilities from a classifier are usable.

- **Pretrained vs Custom Training:** There isn't a widely available pre-trained model specifically for cipher/encoding detection on HuggingFace as of yet, but it is feasible to train one. One strategy, as hinted by the Ciphey tool, is to generate a large corpus of text and encrypt/encode it with various methods to create a labeled dataset ¹⁹. Ciphey's author mentions training a **deep neural network on a corpus (e.g., text from *Harry Potter* novels) encrypted with different algorithms**, to predict the cipher type from the ciphertext ²⁰. This yields a model that can say, for example, *"there's an 81% chance this text is encoded with Base64, 15% chance it's a Vigenère cipher, and 4% chance something else"*. Such a model can be embedded in the microservice to provide a second opinion when heuristics aren't definitive. If open-source, one might start with a smaller transformer like DistilBERT or an LSTM model for quicker inference, and fine-tune on a synthetic dataset covering the target ciphers/encodings. Notably, **Ciphey's approach** uses a custom DNN classifier to rank possible decryptions and is reported to solve most encryptions/encodings in under 3 seconds by pairing the model with guided brute force ²¹ ²⁰.

Resource Considerations: ML models range from lightweight to heavy. A simple CNN or LSTM with a few layers might be only a few megabytes and run inference in milliseconds on CPU – suitable for on-premise or edge deployment. Transformers (BERT-base has 110M parameters) are heavier; on CPU such a model might take hundreds of milliseconds or more for one input, which could be a bottleneck if the service is high-throughput. Options to mitigate this include using a smaller model (DistilBERT has ~66M params), model quantization, or running on a GPU. If deploying in the cloud or on a powerful server, a larger model could be feasible, while for a small local service you might choose a compact model or only call the ML step when necessary. Importantly, because the microservice should work in **near-real-time**, any deep model used should be optimized for inference (possibly using ONNX runtime or similar acceleration).

Accuracy vs Feasibility: In summary, **transformer-based classifiers offer the highest accuracy** in classifying cipher/encoding types (as evidenced by recent research) ²², but they come with higher computational cost. **CNNs or RNNs** provide a middle ground, potentially still very accurate on the specific domain if trained well, and easier to run locally. And **classical ML (SVM/Random Forest)** with good features can be extremely fast, though one must ensure the features cover all relevant cases. Depending on the platform's constraints, one could even deploy multiple models: e.g., a quick heuristic/feature check followed by a more powerful model if needed.

3. Hybrid Approaches (Combining Heuristics and ML)

The most robust systems in practice use **hybrid approaches**, leveraging the strengths of both deterministic rules and statistical learning. There are two main ways to combine them:

- **Ensemble/Stacked Models:** This is where you literally combine outputs of different methods. A notable example is an academic study that took the best feature-engineered classifier and the best neural network, and made an ensemble. For 55 classical ciphers, the **ensemble of a neural network (Transformer) with a feature-based method achieved 82.78% accuracy, significantly better than either alone** ¹⁸. The feature-based component might capture indicators that the neural net missed, and vice versa. In practice, one could implement a meta-classifier that considers both heuristic detections and ML outputs to make a final decision. For instance, if regex strongly indicates

“hex encoding” but the ML model is 60% leaning “Base64”, a logic could favor the regex (perhaps overriding the ML unless the ML confidence is much higher). The ensemble could also be a weighted scheme: e.g., produce a combined confidence score that incorporates heuristic evidence (like a penalty if entropy is too high for plain text, a boost if a decode yields a valid English string, etc.) alongside the ML probability. Designing this requires testing, but it can substantially improve accuracy and reduce false guesses.

- **Sequential (Pipeline) Combination:** This is the approach taken by tools like CyberChef’s Magic and Ciphey, and is very suitable for a microservice that needs to peel away multiple layers. In this approach, **heuristics handle the first pass and obvious layers, and ML kicks in for what remains unresolved**. For example, the microservice might first run a suite of regex checks and quickly detect the input is Base64. It can then decode the Base64 to bytes and see what’s next. Suppose the decoded bytes start with 0x1F8B – the Gzip magic number – the heuristic now knows a compression layer is present ². It would report these two layers (“Base64 -> GZIP compression”) with high confidence. Now, it can decompress the Gzip; if the result is readable text, we’re done. If the result still looks random, we likely have an encrypted layer. At this point, heuristics might not clearly tell if it’s (say) AES or ChaCha20 or some proprietary stream cipher – this is where an ML classifier can analyze the byte pattern of that final blob and predict, for instance, “this looks like AES-CBC encrypted data” with a certain confidence. The service could then output that as the likely encryption type. This **recursive strategy** mirrors how CyberChef Magic performs *speculative execution* of each detected operation and continues searching deeper layers ¹¹. It’s essentially a depth-first search guided by heuristics and checks at each step.
- **Guided Brute-Force with ML Ranking:** Another hybrid strategy is used by **Ciphey**. Ciphey’s pipeline: it uses a **DNN to guess the most likely cipher/encoding**, then **attempts to decode/decrypt in that order**, using a language checker to verify success ²⁰ ²³. For example, if the DNN says “80% chance this is ROT13, 20% chance Base64”, Ciphey will try a ROT13 decode first; if the output appears to be English (its language checker confirms plaintext) ²⁴ ²⁵, it stops and returns the result. If not, it tries Base64 next, and so on. This is a **practical hybrid**: the ML model narrows down the search space so that brute-force (or trying many decoding operations) becomes feasible, and the heuristic language test ensures false positives are filtered out. Such an approach is highly relevant to a CTF context because you often might have to try multiple decryptions. In a microservice, we can integrate this idea by using ML not just for final labeling, but to **prioritize which decoders/solvers to invoke**. For instance, an ML prediction of “likely XOR cipher” could trigger an XOR key search (perhaps using a smaller heuristic tool) to actually decipher the text, whereas a prediction of “AES” might prompt asking the user for a key or trying a known default key if applicable. While implementing full decryption is beyond just *classification*, a ranked suggestion list from ML can greatly optimize any downstream automated solving.

Why Hybrid: Hybrid approaches tend to yield the **highest accuracy and coverage**. Heuristics excel at things that are well-known or easy to detect (giving essentially 100% precision on those, e.g., you won’t mis-identify Base64 if the regex is strict). ML excels at catching patterns that aren’t black-and-white (like distinguishing different high-entropy ciphers, or recognizing a text that’s been XORed by analyzing residual patterns). By combining them, you mitigate each side’s weaknesses. The 2021 study’s result of ~83% accuracy with an ensemble is telling – neither pure neural nets nor pure expert features alone got that high ²⁶. Likewise, the success of CyberChef Magic in real-world usage stems from combining pattern rules with statistical validation ²⁷ ⁵, and Ciphey’s success comes from blending AI with rule-based post-checks ²⁰.

For deployment, a hybrid system allows **performance tuning**: e.g., you can decide to only invoke the heavy ML model if none of the quick heuristics match, which is likely a minority of cases. This keeps average response times low. You can also cache or short-circuit processing if a certain layer is confidently identified (e.g., “we recognized this as gzip, no need to run the ML on raw data that is clearly compressed – first decompress, then continue”).

Recommendation: Use heuristics as a first filter and for recursive decoding of obvious layers, and incorporate an ML model as a backstop for ambiguous cases or to rank deeper decryption attempts. The combination will be powerful: heuristics ensure speed and precision on known formats, while ML provides adaptability and breadth, increasing the system’s overall accuracy.

4. Datasets for Training and Evaluation

To train an ML model or even to fine-tune heuristic thresholds, you’ll need data – examples of various ciphertexts and encodings, ideally labeled with their true type (and possibly the plaintext for verification). In the realm of cryptographic format detection, several dataset sources and strategies are available:

- **Synthetic Data Generation:** This is the most common approach. Take a large collection of sample plaintexts (for example, an English text corpus or randomly generated strings for binary data) and then **encipher or encode them with all the algorithms of interest**. For classical ciphers, one can use literature or Wikipedia text and apply Caesar shifts, Vigenère with random keys, Playfair, etc., generating thousands of samples of each. For modern ciphers, take random keys and IVs to encrypt either random data or real text. Similarly, produce encoded data by applying Base64, hex, URL encoding, etc., to various inputs. The result is a labeled dataset where you know, *this sample = output of AES*, that sample = *Base64 encoding*, and so on. Researchers often generate millions of samples this way to train deep models ¹⁷. For example, Leierzopf et al. generated **10 million ciphertexts of 55 types** (all 55 ACA classical ciphers) for training a classifier ²⁸. Generating data ensures you can have as much as needed, but be careful to use varied plaintexts to avoid bias (you don’t want your model to accidentally learn the quirks of a single book’s text rather than the cipher itself!). Including multi-layer samples in generation can also help if you want the model to detect that (though a recursive heuristic approach might handle multi-layer without needing the model to directly label combinations).
- **Random-Crypto Benchmark:** **Random-Crypto** is an open-source procedurally generated dataset specifically designed for cryptographic CTF challenges ²⁹ ³⁰. It covers *50 different algorithm families and over 5,000 unique tasks* ³⁰. This means it includes a broad range of encryption and encoding scenarios (from classic ciphers to modern encryption, hashing challenges, steganography, etc.), each embedded in a pseudo-realistic challenge format. While Random-Crypto was developed to train reinforcement learning agents, its data could be repurposed for evaluating your detection microservice. For example, you could take the challenge inputs and known solutions from Random-Crypto tasks to see if your detector correctly identifies the methods used. It’s a rich resource because it reflects exactly the kind of data one might see in CTFs – often multi-layered or slightly obfuscated. The Random-Crypto project is open-source, and the paper reports that it provides “*a procedurally generated cryptographic CTF dataset*” with essentially unlimited samples ³¹ ³². You can incorporate this by generating a large number of challenges and extracting the ciphertext parts for training a model, or by directly using it as a test benchmark to measure success rates of the microservice’s guesses.

- **CTF Archives and Repositories:** Beyond Random-Crypto, many past CTF challenges (picoCTF, CSICTF, etc.) have public write-ups and datasets. One could compile a list of ciphertexts from those (with their known type or solution) to build a test set. This ensures your evaluation includes real-world examples, which might be trickier than perfectly generated ones. For instance, a challenge might mix an encoding with a subtle variation (like a custom Base64 alphabet), which a naive regex might miss – testing on such data will highlight if your system needs an update (CyberChef’s Magic, for example, even accounts for **non-standard Base64 alphabets** with multiple regex variants ³³).
- **Language Data for Validation:** If you plan to use language detection (English, etc.) as part of your pipeline, having a “plaintext corpus” is useful. The **English Wikipedia dump** (or any large text corpus) was used by GCHQ to derive byte frequency tables ³⁴ ; you can similarly use it to simulate plaintext outputs. Also gather some non-English plaintext if you expect to encounter those (CyberChef Magic can check against top 38 languages ³⁵). This helps ensure your plaintext detection isn’t English-biased if, say, the flag or message is in another language.
- **Datasets for Hashes and Misc:** If you want to recognize hashes or specific tokens (not necessarily decrypt them, but label them), you can use existing databases. For example, have a list of known hash outputs (many can be generated or taken from places like the Hashcat example files). Similarly, for things like JWT or ASN.1 structures (like X.509 certificate blobs), gather a few examples to test your detector’s heuristic or ML recognition of those formats.

Evaluation Approach: Once you have a dataset, you’d train any ML model on a training portion and then test it on a hold-out set. But equally important is to **evaluate the end-to-end microservice on realistic scenarios**. That means feeding in inputs and checking if the output suggestions are correct (and with high confidence). You should measure metrics like accuracy of top-1 prediction for single-layer inputs, and success rate of fully unraveling multi-layer inputs. If using Random-Crypto or CTF archives, measure how often the microservice’s suggestion matches the actual needed operation(s) to get the plaintext.

By leveraging these datasets, you can ensure the system is well-tuned. For open-source collaboration, consider releasing any corpus you generate (as the Random-Crypto authors did) – it could help others tackling similar problems.

Deployment and Integration Considerations

Lightweight vs. Heavy Models: It’s important to balance accuracy with performance in a microservice context. **Heuristic methods are very lightweight** – they are essentially string operations and basic math, which run quickly even on modest hardware. These should form the first layer of the service and can handle the majority of inputs without any heavy computation. **Machine learning models**, especially deep learning, will require more resources. If the platform allows GPU acceleration (e.g. deploying in a cloud with GPU instances or a local server with a GPU), a transformer model can be served with low latency. If GPU is not an option, consider using smaller models or optimizing with techniques like ONNX runtime or distillation. It might also be viable to run the ML part asynchronously: e.g., the service could return immediate heuristic findings, and separately a refined analysis when the model finishes – but for simplicity, it’s usually better to do a synchronous analysis within a reasonable time (aim for under a second per input for real-time usage). In practice, many inputs will be resolved by heuristics in milliseconds, and only complex cases invoke the model. This **adaptive complexity** keeps the average response fast.

Scalability: Make sure the microservice can handle multiple requests. If using an ML model, loading it into memory once and reusing it (rather than re-loading per request) is important. Frameworks like TensorFlow Serving or PyTorch with TorchServe could be considered, but for a smaller scope, even a simple Flask API that caches the model and processes input might suffice. If high throughput is expected, you might allocate separate worker processes for the ML tasks.

Integration of Tools: You don't have to implement everything from scratch. You can integrate open-source libraries directly: - *CyberChef*: GCHQ provides CyberChef as a web app, but it also has a Node.js API and even a command-line interface. The "Magic" function itself could be called via CyberChef's JavaScript if you embed it, but that might be overkill. Instead, you can replicate its techniques in your service (as outlined above). CyberChef is a great reference for the transformations – you might use its implementations of decoders (e.g., base64 decoder, gzip decompressor, etc.) to actually perform the decoding layers once identified. - *PyWhat/LemmeKnow*: **PyWhat** (Python) or its faster Rust counterpart **LemmeKnow** can be used as a component to identify patterns. For example, PyWhat can tell you if a string is a possible Base64, or a private key, or an IPv6 address, etc., by matching against its extensive regex database ³⁵. Including it as a dependency could jump-start your pattern matching stage. The Rust version, LemmeKnow, claims to be much faster (33x faster) ³⁶, which could be useful if scanning very large inputs or large numbers of patterns. - *Ciphey*: While Ciphey is more of a full tool than a library, parts of it (like its use of the `pywhat` library and its language checker logic) could inspire your implementation. Ciphey's source code on GitHub reveals how it implements the chi-squared and dictionary checks, as well as how it orchestrates multiple decryption attempts. You might not integrate Ciphey wholesale, but you can reuse the concept of its "**Cipher Detection Interface**", which is essentially an ML-based guesser, and its "**Language Checker Interface**", which is the plaintext validator ³⁷. - *Libraries for Crypto*: If you plan for the microservice not just to detect but possibly to attempt decoding/decryption when possible, you can leverage libraries like **PyCryptodome** (for AES, RSA operations), or specialized CTF tools like **cryptanalysis libraries for classical ciphers** (e.g., `simpleSubCrypto` for substitution cipher solving). This goes beyond detection into solving, but it's something to consider for extending the service (as Vitruvian Cipher platform might eventually want automated solving).

Output Format (JSON): Clarity and consistency in output is key for the microservice to be consumed by other parts of the platform. A JSON structure is recommended. It should include: the **detected type(s)**, **confidence scores**, and possibly an indication of next steps or the state of the data after each step. The output needs to handle cases of single-layer vs multi-layer. For example, an output could be:

```
{
  "input_sample": "<original input string>",
  "detected_layers": [
    { "type": "Base64 Encoding", "confidence": 0.99 },
    { "type": "GZIP Compression", "confidence": 0.95 },
    { "type": "AES-128-CBC Encryption", "confidence": 0.80 }
  ],
  "layer_count": 3,
  "final_state": "encrypted",
  "recommendation": "AES decryption (key likely required)"
}
```

In this hypothetical case, the service determined the data was Base64-encoded (almost certain), inside was a Gzip-compressed blob (very likely), and the decompressed result appears to be AES-CBC ciphertext (quite likely). It lists each layer in order. We include a `final_state` field to indicate that after removing those layers, the data is still encrypted (i.e., not yet plaintext). A `recommendation` or `next_action` field can suggest what to do next – here, that an AES decryption is needed, for which a key must be obtained or brute-forced. If the final layer were plaintext, `final_state` might be “plaintext” and the service could even return a snippet or indication of readability. If the service itself doesn’t perform the decryption, it basically hands off the identification (which is the scope of this question). The **confidence scores** help upstream components decide, for instance, if a suggestion should be shown to the user or if the system should auto-apply a decoding. (E.g., if 0.99 confidence Base64, the service might automatically decode that layer; if something is only 0.5 confidence, maybe flag it as a guess.)

For a simpler single-layer input, say the input was just a 32-hex-character string, the output might be:

```
{
  "input_sample": "5d41402abc4b2a76b9719d911017c592",
  "detected_layers": [
    { "type": "MD5 Hash (hex)", "confidence": 0.98 }
  ],
  "layer_count": 1,
  "final_state": "hash",
  "recommendation": "Hash cracking (compare against known hashes or brute-force)"
}
```

This tells us the string looks like an MD5 hash (hex-encoded) with high confidence. The system knows a hash isn’t reversible by decoding, hence `final_state` “hash” and perhaps a recommendation to crack it if needed (which might involve a different service or user input).

The **microservice design** would involve an API (e.g., a REST endpoint `/detect`) where a client sends the data (probably JSON with the string or bytes, since binary may need base64 encoding in transit itself). The service then orchestrates the detection pipeline as described: heuristic checks -> (decode layer and repeat) -> ML classification if needed -> assemble results. Logging each step internally is useful for debugging (and could even be returned if a verbose mode is enabled).

Real-Time Operation: The pipeline as proposed (heuristics first, ML second, recursive decoding) is efficient enough for near-real-time use. Each regex or entropy calculation is microseconds. Decoding Base64 or decompressing gzip is also very fast relative to network and I/O delays. The heaviest part is the ML model, which you can design to be optional. In many cases (especially for typical encodings or easy ciphers in CTFs), the heuristics will crack it without invoking the model. When the model is invoked, it might add, say, tens of milliseconds (with a small model on CPU) up to a few hundred (with a larger model on CPU). If that’s acceptable, you’re set; if not, one can limit the input length sent to the model (e.g., use at most first 1000 bytes or 100 characters, since very long inputs may not need full processing to identify pattern). Transformers in particular have a context length limitation (often 512 tokens by default for BERT), so extremely long inputs might be truncated or sampled for classification. This should be noted in documentation to users.

In terms of **accuracy and maintenance**, a hybrid system might need periodic updates – e.g., if a new encoding becomes popular in CTFs (say, some exotic base91 or a custom variant), you'd add a regex or extend the ML training data to include it. Keeping an eye on false negatives/positives from real usage will guide refinements. But with the methods recommended – regex for common formats, statistical checks, and a well-trained model for the rest – the system will cover a wide range of scenarios.

Conclusion

For the Vitruvian Cipher platform's ciphertext detection microservice, the **recommended solution is a hybrid approach** that **maximizes accuracy while remaining deployable in real-time**. Heuristic detectors (pattern matching, magic bytes, entropy and frequency analysis) provide immediate, high-precision identification of known encodings and compressions ²⁷ ⁵. On top of this, a machine learning classifier (e.g. a fine-tuned Transformer or a lightweight CNN/RNN model) can classify more subtle ciphertexts and provide a probability-based output ¹⁶ ²⁰. By combining the two, either via an ensemble or sequentially (as CyberChef's and Ciphey's designs have shown), we achieve a system that is both **comprehensive and efficient**.

Open-source tools like CyberChef (for its recipes), PyWhat/LemmeKnow (for pattern libraries) ³, and Ciphey (for its integration of ML and heuristics) ¹⁹ ²³ can greatly accelerate development. Suitable datasets such as Random-Crypto ³⁰ and generated corpora of encrypted text will ensure the ML component is well-trained and the system is evaluated on relevant challenges. The output will be a clear JSON report of detected layers and confidences, enabling downstream automation (for example, automatically applying a decoding, or alerting a user that "This looks like an XOR cipher").

By following these recommendations, the microservice will essentially replicate and even enhance CyberChef Magic's functionality in an automated fashion – **suggesting decoding/decryption layers with confidence scores, in real-time**, and ready to integrate into the Vitruvian Cipher platform's microservice architecture. Such a system will significantly aid CTF players or automated agents in quickly recognizing how a secret message is concealed, bridging the gap from receiving a jumbled string to knowing which tools or keys to apply next in order to reveal the hidden flag.

Sources:

- GCHQ CyberChef documentation on the "Magic" automatic encoding detection ²⁷ ³⁸
- Sikdar & Kule (2024), *Ciphertext identification using CNN/Transformers/BERT* – notes on model efficacy ¹⁶
- Leierzopf et al. (2021), *Detection of Classical Cipher Types* – ensemble approach accuracy ²⁶
- Ciphey project blog – hybrid strategy with DNN and chi-squared language check ²⁰ ⁹
- PyWhat library – regex patterns for common encodings/hashes (identifies Base64, Base85, etc.) ³
- Random-Crypto benchmark (2025) – procedurally generated CTF crypto dataset covering 50 algorithms ³⁰

¹ ² ⁴ ⁵ ⁶ ⁷ ⁸ ¹¹ ¹² ²⁷ ³³ ³⁴ ³⁵ ³⁸ Automatic detection of encoded data using CyberChef Magic · gchq/CyberChef Wiki · GitHub

<https://github.com/gchq/CyberChef/wiki/Automatic-detection-of-encoded-data-using-CyberChef-Magic>

3 Python Secrets Revealed with PyWhat | by Py-Core Python Programming | Python in Plain English

<https://python.plainenglish.io/python-secrets-revealed-with-pywhat-ac61da42581a?gi=457c3a5948d8>

9 10 19 20 23 24 25 Ciphey - Automated Decryption Tool

<https://skerritt.blog/ciphey/>

13 GitHub - HappyHackingOrange/Ciphertext-Classification-Ensemble-SVM: This is a demonstration of performing ensemble learning with support vector machine (SVM) to classify type of cipher used to encrypt texts.

<https://github.com/HappyHackingOrange/Ciphertext-Classification-Ensemble-SVM>

14 15 16 22 Intelligent Identification of Cryptographic Ciphers using Machine Learning Techniques

<https://www.mecs-press.org/ijisa/ijisa-v16-n6/v16n6-2.html>

17 18 26 28 cryptool.org

https://www.cryptool.org/download/ncid/Detect-Classical-Cipher-Types-with-Feature-Learning_AusDM2021_PrePrint.pdf

21 Ciphey - HackDB

<https://hackdb.com/item/ciphey>

29 30 31 32 Improving LLM Agents with Reinforcement Learning on Cryptographic CTF Challenges

<https://arxiv.org/html/2506.02048>

36 Introducing Ares - The Fastest Way to Decode Anything - Skerritt.blog

<https://skerritt.blog/introducing-ares/>

37 ciphey: Automatically decrypt encryptions without knowing the key or cipher, decode encodings, and crack hashes

https://gitee.com/ajax2018/ciphey?skip_mobile=true