

Forum:[Hack Forums](#)

Thread: [Aetherflow v5 - 100TB of raw data can be compressed into a single **~350MB**](#)

SIM-5724601 - January 21, 2026, 9:19 pm

Aetherflow

A high-density data abstraction protocol capable of mapping massive datasets into symbolic representations.

Description

Aetherflow implements a cutting-edge data abstraction layer. It is designed to map entire data centers into single text files by utilizing 1MB "Super-Planes" mapped to single Unicode code points in the Private Use Area.

Performance & Compression

Aetherflow v5 sets the industry standard for density and speed:

- Massive Compression: 100TB of raw data can be compressed into a single ~350MB text file (approx. 333,333:1 ratio).
- Extreme Velocity: Capable of processing data at petabyte-per-hour speeds using recursive symbolic reconstitution.
- Symbolic Mapping: Every 1MB of data is represented by a single unique Unicode character.

Features

- AetherFlow v5 Protocol: Petabyte-scale abstraction.
- Data Center Mapping: High-density symbolic reconstitution.
- Trial Service: Free for development and testing.

Donations

If you find this protocol useful and want to support its continued development, donations are welcome:

- Bitcoin (BTC): `1MmZZCwx7HPWBBgHdL9tw52JFyEmgUDBgo`

License

This project is provided as a trial service for development. Future commercial licensing terms will apply for production environments.

<https://github.com/sam-altman-dev/aetherflow>

LiNixx - January 21, 2026, 9:45 pm

Ermeee I'm skeptical. This is mathematically and physically impossible.

SIM-5724601 - January 21, 2026, 10:07 pm

(Jan 21, 2026 10:45 PM)LiNixx Wrote: [\[CITE\]](#) Ermeee I'm skeptical. This is mathematically and physically impossible.

Benchmark file in repository demos it.

LiNixx - January 22, 2026, 1:25 am

Encoding 1 MB of data into a single Unicode character is mathematically impossible without losing virtually all the information. 1 MB is 8,388,608 bits A Unicode character can represent numbers

from 0 up to 1,114,111.

That means it can hold about 21 bits of information

You can't stuff over 8 million bits of information into 21 bits, you just cant.

Your “benchmark” is just compressing 1 MB of repeated AAAAAAAAAA... the absolute easiest case. Most real-world data has far higher entropy and will not compress anywhere near those claimed ratios. Then you extrapolate that to Hundreds of Terabytes of data? Guy that's impossible. The data would be corrupted beyond recognition.

Looking at your code:

```
Code for (let i = 0; i < input.length; i += SUPER_STRIDE) {  
  const end = Math.min(i + SUPER_STRIDE, input.length);  
  let stateSum = 0;  
  for (let j = i; j < end; j++) {  
    stateSum = (stateSum + input[j] * ((j - i) % 31)) % 0xFFFF;  
  }  
  hgStream += String.fromCodePoint(HG_OFFSET + stateSum);  
}
```

This is not compression. It's a checksum/hash of each 1 MB block encoded as a Unicode glyph. A checksum is lossy by design. The “decompression” step just generates made up bytes from that checksum.

If anyone tried this on a large real dataset, I'm almost positive it would destroy the data. That's probably why the “benchmark” only tests a single MB and then extrapolates unrealistically.

Everybody should be highly skeptical of these claims. Maybe some staff or someone more experienced can take a deeper look, but from what I see, this is not a real compression algorithm.

7 - January 22, 2026, 4:54 pm

Sorry but AI says

--

Not possible as described (not losslessly, not with those ratios), and the “Unicode 1MB -> 1 character” claim gives it away immediately.

Why it's impossible (in simple terms)

A 1MB block contains:

1 MB = 1,048,576 bytes

= 8,388,608 bits

A single Unicode code point (even using the full Unicode range) can only represent about:

~1,114,112 possible values

that's about $\log_2(1,114,112)$? 20–21 bits of information

So you're trying to store 8.3 million bits inside ~21 bits.

That cannot be done without losing essentially all the information.

This isn't "industry leading compression" — it violates the basic rules of information theory (Shannon entropy / lossless compression limits).

What it actually is

From the code shown in the thread, it's not compression at all — it's basically:

- ? taking each 1MB chunk
- ? computing a tiny checksum / hash-like summary (16-bit-ish range)
- ? encoding that number into a Unicode character
- ? throwing away the original data

Then "decompression" can only generate fake bytes from the checksum, because the original information is gone.

So it's lossy irreversible mapping, not compression.

The "350MB for 100TB" claim

That ratio (333,333:1) would only be possible if the 100TB dataset had almost no entropy (like trillions of repeating bytes: AAAAAAAAAAAAAA...), which is not what real data looks like.

Real datasets contain:

already-compressed files (zip, mp4, jpg, pdf)

encrypted data

databases with high entropy sections

Those do not compress meaningfully beyond a small %.

Verdict

- ? Possible as a "fingerprint / sketch / signature" system (like block hashes)
- ? Not possible as lossless compression
- ? Not possible to restore a real dataset from it
- ? Not possible at the claimed scale and speed

So: it's marketing + misunderstanding (or deliberate scammy framing), not a breakthrough.

SIM-5724601 - March 13, 2026, 11:23 am

(Jan 22, 2026 05:54 PM)7 Wrote: [\[CITE\]](#) Sorry but AI says

--

Not possible as described (not losslessly, not with those ratios), and the "Unicode 1MB -> 1 character" claim gives it away immediately.

Why it's impossible (in simple terms)

A 1MB block contains:

1 MB = 1,048,576 bytes

= 8,388,608 bits

A single Unicode code point (even using the full Unicode range) can only represent about:

~1,114,112 possible values

that's about $\log_2(1,114,112) \approx 20$ –21 bits of information

So you're trying to store 8.3 million bits inside ~21 bits.

That cannot be done without losing essentially all the information.

This isn't "industry leading compression" — it violates the basic rules of information theory (Shannon entropy / lossless compression limits).

What it actually is

From the code shown in the thread, it's not compression at all — it's basically:

- ? taking each 1MB chunk

- ? computing a tiny checksum / hash-like summary (16-bit-ish range)

- ? encoding that number into a Unicode character

- ? throwing away the original data

Then "decompression" can only generate fake bytes from the checksum, because the original information is gone.

So it's lossy irreversible mapping, not compression.

The "350MB for 100TB" claim

That ratio (333,333:1) would only be possible if the 100TB dataset had almost no entropy (like trillions of repeating bytes: AAAAAAAAAAAAAA...), which is not what real data looks like.

Real datasets contain:

already-compressed files (zip, mp4, jpg, pdf)

encrypted data

databases with high entropy sections

Those do not compress meaningfully beyond a small %.

Verdict

- ? Possible as a "fingerprint / sketch / signature" system (like block hashes)

- ? Not possible as lossless compression

- ? Not possible to restore a real dataset from it

? Not possible at the claimed scale and speed

So: it's marketing + misunderstanding (or deliberate scammy framing), not a breakthrough.
No possible with today's technology but imagine this the touchscreen cellphones had been a lot larger in 1980's and technology is creating and interchanging. I promise this is legit.

(Jan 22, 2026 02:25 AM)LiNixx Wrote: [\[CITE\]](#) Encoding 1 MB of data into a single Unicode character is mathematically impossible without losing virtually all the information. 1 MB is 8,388,608 bits A Unicode character can represent numbers from 0 up to 1,114,111.

That means it can hold about 21 bits of information

You can't stuff over 8 million bits of information into 21 bits, you just cant.

Your "benchmark" is just compressing 1 MB of repeated AAAAAAAAAA... the absolute easiest case. Most real-world data has far higher entropy and will not compress anywhere near those claimed ratios. Then you extrapolate that to Hundreds of Terabytes of data? Guy that's impossible. The data would be corrupted beyond recognition.

Looking at your code:

```
Code for (let i = 0; i < input.length; i += SUPER_STRIDE) {  
  const end = Math.min(i + SUPER_STRIDE, input.length);  
  let stateSum = 0;  
  for (let j = i; j < end; j++) {  
    stateSum = (stateSum + input[j] * ((j - i) % 31)) % 0xFFFF;  
  }  
  hgStream += String.fromCodePoint(HG_OFFSET + stateSum);  
}
```

This is not compression. It's a checksum/hash of each 1 MB block encoded as a Unicode glyph. A checksum is lossy by design. The "decompression" step just generates made up bytes from that checksum.

If anyone tried this on a large real dataset, I'm almost positive it would destroy the data. That's probably why the "benchmark" only tests a single MB and then extrapolates unrealistically.

Everybody should be highly skeptical of these claims. Maybe some staff or someone more experienced can take a deeper look, but from what I see, this is not a real compression algorithm. It's not unicode.

Hackers - March 13, 2026, 2:21 pm

I reached out to this guy in his other thread where he claimed he could do 100TB too 100KB and he sent some wack ass script with placeholders and logic that specifically prevented the testing of it... as verification. lol

Checksum comments above are accurate. This tool cannot compress 100TB to even 1TB, let alone 350MB or 100KB. I would be surprised if he could even compress 100TB to 10TB. OP is free to prove me wrong, but he can't, so... Yeah, nothing else to say. The thread's a waste of disk space on Omni's server. Like the anti-compression, he created unnecessary data to talk about something he can't even do.

This is some bullshit and since we're not in marketplace, I think we can say OP would be scamming if he tried to sell this.

SIM-5724601 - March 13, 2026, 3:04 pm

(Mar 13, 2026 02:21 PM)Hackers Wrote: [\[CITE\]](#) I reached out to this guy in his other thread where he claimed he could do 100TB too 100KB and he sent some wack ass script with placeholders and logic that specifically prevented the testing of it.... as verification. lol

Checksum comments above are accurate. This tool cannot compress 100TB to even 1TB, let alone 350MB or 100KB. I would be surprised if he could even compress 100TB to 10TB. OP is free to prove me wrong, but he can't, so... Yeah, nothing else to say. The thread's a waste of disk space on Omni's server. Like the anti-compression, he created unnecessary data to talk about something he can't even do.

This is some bullshit and since we're not in marketplace, I think we can say OP would be scamming if he tried to sell this.

<https://gist.github.com/general-satan/0c9655f3c104ba7cf1a2a0fea3a7fabf>

"So it's like 7zip but 100TB God?"

Kind of, but with a critical difference:

7-Zip actually compresses your data — you get the original file back byte-for-byte on extraction. It finds patterns and encodes them smaller, but it's still bounded by physics. A truly random 100TB file would compress to roughly 100TB.

AetherFlow doesn't compress your data at all — it discards it and saves only the recipe (the AES key + IV). On "unzip" it regenerates a deterministic data stream from that recipe, not your original file. The 312-byte seed and your 100TB file are completely unrelated in content.

So it's less "7-Zip but bigger" and more like saying:

"I compressed the entire Library of Congress down to a single coordinate on a map. To 'restore' it, I redraw the map from that coordinate."

The restored data is consistent and verifiable (the block CRC matches every time), but it's not your 100TB — it's the AES engine's output for those parameters. The original is gone.

Real-world compression tools like 7-Zip, Zstd, or LZMA are bounded by Shannon's information theory — you can't losslessly compress truly random data below its entropy. AetherFlow sidesteps that entirely by not preserving the original, which is why the "100TB -> 312 bytes" claim needs that asterisk.

Null State - March 13, 2026, 5:29 pm

The program doesn't actually compress anything.

What it does is read a file, calculate its size and CRC32 checksum, and then write a small metadata file (`.aef`) containing:

- a magic ID
- a fixed key and IV
- the original file size
- the file's CRC32 checksum
- * the original filename

The resulting seed file is only a few hundred bytes because it doesn't store the original data at all. It's essentially a manifest or fingerprint, not a compressed archive.

The AES-NI code also isn't performing real encryption or reconstruction of the file. It just generates a small deterministic block used for a demonstration checksum.

So the claim that it can compress 100 TB down to 350 MB is misleading. The program simply records metadata about a file. If the original data is deleted, the `.aef` file cannot recreate it.

In other words, it's closer to a file integrity/verification manifest than a compression system.

SIM-5724601 - March 13, 2026, 5:44 pm

(Mar 13, 2026 05:29 PM)Null State Wrote: [\[CITE\]](#) The program doesn't actually compress anything.

What it does is read a file, calculate its size and CRC32 checksum, and then write a small metadata file (`.aef`) containing:

- a magic ID
- a fixed key and IV
- the original file size
- the file's CRC32 checksum
- * the original filename

The resulting seed file is only a few hundred bytes because it doesn't store the original data at all. It's essentially a manifest or fingerprint, not a compressed archive.

The AES-NI code also isn't performing real encryption or reconstruction of the file. It just generates a small deterministic block used for a demonstration checksum.

So the claim that it can compress 100 TB down to 350 MB is misleading. The program simply records metadata about a file. If the original data is deleted, the `.aef` file cannot recreate it.

In other words, it's closer to a file integrity/verification manifest than a compression system. Did you try zip and unzip commands on a file?

Null State - March 13, 2026, 7:41 pm

(Mar 13, 2026 05:44 PM)SIM-5724601 Wrote: [\[CITE\]](#) Did you try zip and unzip commands on a file?

No, but here's your revolutionary algorithm re-written. AI commented to break it all down. Code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <array>
#include <stdint>
#include <cstring>
#include <iomanip>
#include <chrono>
#include <immintrin.h>
```

```
/*
```

```
    AetherFlow Seed Header
```

```
    This structure is the on-disk metadata format.
```

```
    It stores:
```

```
    - a magic identifier
```

```

- a 16-byte key
- a 16-byte IV
- original file size
- CRC32 of the original file
- original source name
*/
struct SeedHeader
{
    uint64_t magic_id;
    uint8_t key[16];
    uint8_t iv[16];
    uint64_t original_size;
    uint32_t original_crc32;
    char source_name[256];
};

/*
    FileStats holds the results of scanning a file.
*/
struct FileStats
{
    uint64_t size = 0;
    uint32_t crc32 = 0;
    bool ok = false;
};

/*
    CRC32 calculator.

    This is a small stateful CRC32 helper so we can
    feed bytes incrementally from memory or files.
*/
class CRC32
{
public:
    CRC32()
        : value_(0xFFFFFFFFu)
    {
    }

    /*
        Feed one byte into the CRC state.
    */
    void update_byte(uint8_t byte)
    {
        value_ ^= byte;

        for (int i = 0; i < 8; ++i)
        {
            value_ = (value_ >> 1) ^ (0xEDB88320u & -(value_ & 1));
        }
    }
}

```

```

/*
    Feed a block of bytes into the CRC state.
*/
void update_buffer(const uint8_t* data, size_t length)
{
    for (size_t i = 0; i < length; ++i)
    {
        update_byte(data[i]);
    }
}

/*
    Return final CRC32 value.
*/
uint32_t final() const
{
    return ~value_;
}

private:
    uint32_t value_;
};

/*
    AetherBlockGenerator

    This intentionally mirrors the same conceptual behavior
    as the original code: one deterministic 16-byte block is
    generated using AES-NI and an incrementing state.

    This is not a real encryption mode. It is only a deterministic
    block generator used for the program's metadata demo behavior.
*/
class AetherBlockGenerator
{
public:
    AetherBlockGenerator(const uint8_t* key, const uint8_t* iv)
    {
        key_ = _mm_loadu_si128(reinterpret_cast<const __m128i*>(key));
        state_ = _mm_loadu_si128(reinterpret_cast<const __m128i*>(iv));
    }

    /*
        Generate one 16-byte block into the caller-provided buffer.
    */
    void next_block(uint8_t out[16])
    {
        __m128i generated = _mm_aesenc_si128(state_, key_);

        /*
            Increment the internal 128-bit state as two packed 64-bit lanes,
            same idea as the original code.
        */
    }

```

```

    state_ = _mm_add_epi64(state_, _mm_set1_epi64x(1));

    _mm_storeu_si128(reinterpret_cast<__m128i*>(out), generated);
}

private:
    __m128i key_;
    __m128i state_;
};

/*
    Return CRC32 for a raw memory block.
*/
uint32_t crc32_of_memory(const uint8_t* data, size_t length)
{
    CRC32 crc;
    crc.update_buffer(data, length);
    return crc.final();
}

/*
    Scan a file and return size + CRC32.

    This reads the file in chunks so large files do not have to
    fit into memory.
*/
FileStats scan_file(const std::string& path)
{
    FileStats result;

    std::ifstream file(path, std::ios::binary);
    if (!file)
    {
        return result;
    }

    constexpr size_t CHUNK_SIZE = 1024 * 1024;
    std::vector<uint8_t> buffer(CHUNK_SIZE);

    CRC32 crc;

    while (file.read(reinterpret_cast<char*>(buffer.data()), static_cast<std::streamsize>(buffer.size())))
    ||
        file.gcount() > 0)
    {
        const size_t bytes_read = static_cast<size_t>(file.gcount());

        result.size += static_cast<uint64_t>(bytes_read);
        crc.update_buffer(buffer.data(), bytes_read);
    }

    /*
        If the stream failed for a reason other than EOF, reject it.
    */
}

```

```

    */
    if (file.bad())
    {
        return FileStats{};
    }

    result.crc32 = crc.final();
    result.ok = true;
    return result;
}

/*
    Extract just the last path element from a full path string.

    This avoids needing <filesystem> and keeps the program
    self-contained.
*/
std::string basename_only(const std::string& path)
{
    const size_t slash_pos = path.find_last_of("/\\");
    if (slash_pos == std::string::npos)
    {
        return path;
    }

    return path.substr(slash_pos + 1);
}

/*
    Print command usage.
*/
void print_usage()
{
    std::cout
        << "AetherFlow Engine v1.0\n"
        << "Usage:\n"
        << "  aetherflow zip  <input_file>\n"
        << "  aetherflow unzip <input_file.aef>\n";
}

/*
    Fill a SeedHeader with deterministic metadata values.

    This preserves the same style as the original:
    - same magic
    - deterministic key bytes
    - deterministic IV bytes
*/
SeedHeader build_seed_header(const std::string& input_path, const FileStats& stats)
{
    SeedHeader header{};
    header.magic_id = 0x41455448ULL;
    header.original_size = stats.size;

```

```

header.original_crc32 = stats.crc32;

for (int i = 0; i < 16; ++i)
{
    header.key[i] = static_cast<uint8_t>(i * 7);
    header.iv[i] = static_cast<uint8_t>(i + 3);
}

const std::string name_only = basename_only(input_path);

/*
    Copy filename with truncation if needed, leaving a null terminator
    because the struct was zero-initialized above.
*/
const size_t copy_len = (name_only.size() < 255) ? name_only.size() : 255;
std::memcpy(header.source_name, name_only.data(), copy_len);

return header;
}

/*
    Write one seed header to disk.
*/
bool write_seed_file(const std::string& output_path, const SeedHeader& header)
{
    std::ofstream out(output_path, std::ios::binary);
    if (!out)
    {
        return false;
    }

    out.write(reinterpret_cast<const char*>(&header), sizeof(header));
    return static_cast<bool>(out);
}

/*
    Read one seed header from disk.
*/
bool read_seed_file(const std::string& input_path, SeedHeader& header)
{
    std::ifstream in(input_path, std::ios::binary);
    if (!in)
    {
        return false;
    }

    in.read(reinterpret_cast<char*>(&header), sizeof(header));
    return static_cast<bool>(in);
}

/*
    Generate the 16-byte demo block from the seed and return its CRC32.
*/

```

```

uint32_t generate_demo_block_crc32(const SeedHeader& header)
{
    AetherBlockGenerator generator(header.key, header.iv);

    uint8_t block[16];
    generator.next_block(block);

    return crc32_of_memory(block, sizeof(block));
}

/*
    Handle "zip" mode.

    This does not compress data. It creates the tiny metadata seed file,
    just like the original feature set.
*/
int run_zip_mode(const std::string& input_path)
{
    std::cout << "[AetherFlow] Seeding: " << input_path << "\n";

    const auto start = std::chrono::steady_clock::now();

    const FileStats stats = scan_file(input_path);
    if (!stats.ok)
    {
        std::cerr << "[ERROR] Cannot read file: " << input_path << "\n";
        return 1;
    }

    /*
        Preserve original behavior: reject size 0 as an error.
        This matches the feature behavior of the original program.
    */
    if (stats.size == 0)
    {
        std::cerr << "[ERROR] Cannot read file: " << input_path << "\n";
        return 1;
    }

    const SeedHeader header = build_seed_header(input_path, stats);
    const std::string output_path = input_path + ".aef";

    if (!write_seed_file(output_path, header))
    {
        std::cerr << "[ERROR] Cannot write: " << output_path << "\n";
        return 1;
    }

    const uint32_t block_crc32 = generate_demo_block_crc32(header);

    const auto stop = std::chrono::steady_clock::now();
    const double elapsed_ms =
        std::chrono::duration<double, std::milli>(stop - start).count();

```

```

std::cout
    << "[AetherFlow] Source file : " << input_path << "\n"
    << "      Size      : " << stats.size << " bytes ("
    << std::fixed << std::setprecision(2)
    << (static_cast<double>(stats.size) / 1e6) << " MB)\n"
    << "      CRC32      : 0x"
    << std::hex << std::uppercase
    << std::setw(8) << std::setfill('0') << stats.crc32
    << std::dec << "\n"
    << "      Seed file   : " << output_path << "\n"
    << "      Seed size   : " << sizeof(SeedHeader) << " bytes\n"
    << "      Time        : "
    << std::fixed << std::setprecision(1) << elapsed_ms << " ms\n"
    << "      Block CRC32  : 0x"
    << std::hex << std::uppercase
    << std::setw(8) << std::setfill('0') << block_crc32
    << std::dec << "\n"
    << "[AetherFlow] Zip complete.\n";

return 0;
}

/*
Handle "unzip" mode.

This reads and verifies the seed file metadata, prints details,
generates the same demo block CRC32, and optionally re-checks
the original source file if it still exists.
*/
int run_unzip_mode(const std::string& seed_path)
{
    std::cout << "[AetherFlow] Verifying seed: " << seed_path << "\n";

    const auto start = std::chrono::steady_clock::now();

    SeedHeader header{};
    if (!read_seed_file(seed_path, header))
    {
        std::cerr << "[ERROR] Corrupt, incomplete, or missing seed file: " << seed_path << "\n";
        return 1;
    }

    if (header.magic_id != 0x41455448ULL)
    {
        std::cerr << "[ERROR] Invalid magic ID. Not an AetherFlow seed file.\n";
        return 1;
    }

    const uint32_t block_crc32 = generate_demo_block_crc32(header);

    const auto stop = std::chrono::steady_clock::now();
    const double elapsed_ms =

```

```

std::chrono::duration<double, std::milli>(stop - start).count();

std::cout
    << "[AetherFlow] Seed file    : " << seed_path << "\n"
    << "          Source file    : " << header.source_name << "\n"
    << "          Original size : " << header.original_size << " bytes ("
    << std::fixed << std::setprecision(2)
    << (static_cast<double>(header.original_size) / 1e6) << " MB)\n"
    << "          Stored CRC32   : 0x"
    << std::hex << std::uppercase
    << std::setw(8) << std::setfill('0') << header.original_crc32
    << std::dec << "\n"
    << "          Block CRC32    : 0x"
    << std::hex << std::uppercase
    << std::setw(8) << std::setfill('0') << block_crc32
    << std::dec << "\n"
    << "          Time           : "
    << std::fixed << std::setprecision(1) << elapsed_ms << " ms\n";

/*
If the original source file is still present, re-scan it and compare.
*/
{
    std::ifstream test_source(header.source_name, std::ios::binary);

    if (test_source.good())
    {
        test_source.close();

        const FileStats verify = scan_file(header.source_name);

        const bool match =
            verify.ok &&
            verify.size == header.original_size &&
            verify.crc32 == header.original_crc32;

        std::cout
            << "          Re-verified   : 0x"
            << std::hex << std::uppercase
            << std::setw(8) << std::setfill('0') << verify.crc32
            << std::dec
            << (match ? " [MATCH - Lossless OK]" : " [MISMATCH - Data changed!]")
            << "\n";
    }
    else
    {
        std::cout
            << "          Re-verify    : source file not present (offline verify)\n";
    }
}

std::cout << "[AetherFlow] Unzip complete.\n";
return 0;

```

```

}

int main(int argc, char* argv[])
{
    if (argc < 3)
    {
        print_usage();
        return 1;
    }

    const std::string mode = argv[1];
    const std::string path = argv[2];

    if (mode == "zip")
    {
        return run_zip_mode(path);
    }
    else if (mode == "unzip")
    {
        return run_unzip_mode(path);
    }
    else
    {
        std::cerr << "[ERROR] Unknown mode: " << mode << "\n";
        print_usage();
        return 1;
    }
}

```

Doug Dimmadome - March 14, 2026, 7:18 pm

This post reads like something you generated with AI and the AI just kept telling you, "Oh, you're so smart! Yeah, that'll work!"

EDIT: Yeah, there's a .replit file in the repo. Just AI slop