# Generative AI: A Case Study Highlighting the Potential and the Risks

Translating legacy SAS code to modern open-source alternatives

Nikolay Manchev *

Domino Data Lab

October 13, 2023

**Abstract**

In this paper we look at how the use of permissive-licensed languages like Python and R has been steadily increasing in the pharmaceutical industry, often at the expense of proprietary technologies like SAS. We consider the large amount of technical know-how that is currently implemented in SAS and investigate how with the help of Large Language Models (LLMs) such existing code can be migrated to more open and widely used alternatives like R and Python. We cover some key challenges around source to source translation in detail and provide a blueprint for building a modern translation pipeline, which is flexible, cost efficient, compliant, and produces reproducible results.

## 1 The changing technology landscape

When it comes to statistical analysis and data management, the Statistical Analysis System (SAS) software suite has long dominated the pharmaceutical industry. Some of the reasons, which have historically driven the adoption of SAS, are its maturity, regulatory acceptance, and strong user community. SAS has indeed a long-established presence as the software suite has been around since the 1960s. It is widely accepted by regulatory agencies as the U.S. Food and Drug Administration and the European Medicines Agency, and its popularity in academia has maintained a pool of trained resources that the industry can draw from.

It is worth noting, however, that the technology landscape is rapidly evolving and although SAS has had a dominant position historically, its widespread use has been increasingly challenged by open-source ecosystems centred around languages like R and Python. The 10th annual ranking of the top programming languages published by IEEE Spectrum [1] points out that not only Python remains the No. 1 ranking language in 2023, but it also "widens its lead". In this ranking Python holds first place with the maximum score of 1.0 while R is ranked 11th with a score of 0.1316 and SAS is put in 15th place[1].

*nikolay.manchev@dominodatalab.com

[1]The methodology uses 8 metrics to judge language popularity and provides a normalised score in [0,1] for the first 13 entries. No index value is given for SAS as it currently sits in 15th place. More details on the methodology are available here: `https://spectrum.ieee.org/top-programming-languages-methodology`

One could argue that compared to other industries the changes of the technology stack in the pharmaceutical industry has not been as rapid. This can be attributed to the inherent caution and risk averse attitude imposed by heavy regulation, but the explosion of new frameworks and tools and the wide adoption of data science in other domains is becoming a driving force that is difficult to ignore. Various factors related to intellectual property and trade secrets make quantitative data on use of programming languages in the pharma industry hard to obtain. There is, however, a growing amount of evidence suggesting that the pharmaceutical industry is indeed showing an increasing trend of using R for data analysis and statistical purposes.

DiIorio and Abolafia [2] conducted a survey across 25 industry leaders employed at both pharmaceutical companies and contract research organisations, and provide data from the respondent on the use of programming languages. Table 1 indicates that although the use of SAS remained stable during the period of interest, there is certainly a growing adoption of the use of R.

Table 1: Programming languages used to produce deliverables today compared to 10 or more years ago from DiIorio and Abolafia [2]. Survey respondents is $N = 10$. Note that the paper doesn't specify a timeframe for when the data was collected, but given its publication date we can place the evaluated period around 2008-2018 $\pm$ 1 year.

| **Language** | **10 Years Ago (N)** | **Today (N)** |
|---|---|---|
| SAS | 10 | 10 |
| Some R | 1 | 4* |
| Extensive use of R | 0 | 0 |
| Other (SPSS, JMP) | 1 | 0 |

* Respondents stated that R was used for a "few graphs". Two of the four respondents stated that R was not used for deliverables.

At Domino Data Lab we conducted a similar survey in March 2023 with $N = 15$ respondents from US-based pharmaceutical companies. Each respondent was specifically selected to be in a leadership position in the biostatistics, quantitative science, advanced analytics or related field in their respective company. The results from the survey are given in Table 2 and they paint a remarkable picture — the survey data suggests that across the 15 companies SAS, Python, and R are almost at complete parity. It is also interesting to note that 3 of the companies report no use of SAS whatsoever. Although both studies have been carried out with a relatively low number of participants, there is a clear trend of closing the gap between the use of proprietary SAS technology and more rapidly evolving tools from the R and Python ecosystems.

Table 2: Programming languages used in the organisation. Note, that unlike the DiIorio and Abolafia's data the use of programming languages here is not restricted only to production of deliverables (although DiIorio and Abolafia define this term loosely as "analysis datasets, tables, figures..." so it is not entirely clear if any artefacts need to be specifically excluded.)

| **Language** | **Number of responses** |
|---|---|
| SAS | 12 |
| R | 12 |
| Python | 11 |

# 2 Source-to-source translation

In computer science a source-to-source (S2S) translator (or S2S compiler) is a language processor that takes source code from one programming language and produces an equivalent program in a different programming language. The appeal of using S2S systems is that, if implemented correctly, they can increase the longevity, reliability, and cost efficiency of software projects. Hinchey et al. [3] point out that although hardware dependability has been increasing over the years, the same cannot be said about the reliability of software systems. Indeed, the need to often migrate software between different platforms and underlying infrastructure often mandates a change in programming language and toolset. Adverse effects like scope creep and sprawling legacy code are also drivers for "clean" rewrites.

Unlike traditional compilers that typically translate code between a higher and lower programming languages, S2S translation usually happens at the same abstraction level. Some of the first commercially available S2S compilers were built in the early 1980s for the purposes of translating assembly language source code between different CPUs. Examples include Intel CONV86 [4], a tool to convert error-free 8080/8085 source files to syntactically valid ASM86 source files, and Seattle Computer Products' TRANS86.COM [5], which can translate Intel 8080 and Zilog Z80 assembly source code into Intel 8086 assembly. In addition, proliferation of multi-core CPUs gave rise to a number of translators aiming to rewrite the source code in the same high-level language, but apply optimisations that will make it leverage the available processing units in parallel. One example of such translator is PIPS [6], which can take C or Fortran 77 as input and generate restructured and more optimal version of the original source code.

The early attempts to implement S2S compilers were rudimentary and relied predominantly on text manipulation and pattern matching techniques. The quality of the translation they provided was often poor and their limitations were widely known and well recognised. Here we highlight some frequent issues to equip the reader with better intuition on the challenges associated with automatic code translation.

## 2.1 Unsupported features in the target language

CONV86, for example, has a mechanism named "caution messages", which it uses when it can't handle the ambiguity of the input or is uncertain about the translated output. Caution messages, as implemented in CONV86, do not necessary mean that the output won't function as intended, but they are typically a strong signal that some manual editing or at least human verification is in order. The most basic example that could trigger a caution message would be the use of an 8085-specific instructions that does not have an 8086 equivalent (e.g. Read in Memory (`RIM`) or Set In Memory (`SIM`) instructions).

## 2.2 Undefined language behaviour

Sometimes, the source language specification refers to certain behaviours that are not defined by the standard. Such behaviours are often implementation specific and may result in compiled code that responds differently under identical circumstances. An example of undefined behaviour in the C specification is illustrated in the following code snipped:

```
#include <stdio.h>

int main() {
    int x = 5;
```

```
    int y = 0;

    int result = x / y;  // Division by zero

    printf("Result: %d\n", result);

    return 0;
}
```

According to the C99 standard division by zero is undefined behaviour: "if the value of the second operand is zero, the behavior is undefined" [7]. This means that what happens when you run the code above can vary from one compiler to another. Some compilers may produce an error or a warning, while others may allow the program to run with unpredictable results such as crashing or producing incorrect output.

## 2.3 Inability to recognise intent

As we already established, sometimes a source snippet cannot be translated like-for-like to the target language (be it because of unsupported features in the target language/platform or other limitations). On the other hand, it is also often the case that pattern matching techniques simply fail to recognize the intent behind the source code. In trying to perfectly mimic the source and produce an equivalent functionality, they frequently fail to recognise the purpose of the original material and, by extension, fail to produce a more elegant solution using the specific features of the target language. Let's look at the following line of C code:

```
while (*m++ = *p++);
```

A seasoned C programmer will quickly recognise that this line represents a compact way of copying a string from one location to another location. We can expanded the snipped above into a complete C program as follows:

```
int main() {
    char source[] = "Hello, World!"; // Source string
    char destination[20];            // Destination string

    char *p = source;                // Pointer to source string
    char *m = destination;           // Pointer to destination string

    // Copy the source string to the destination string
    while (*m++ = *p++);

    printf("Source: %s\n", source);
    printf("Destination: %s\n", destination);

    return 0;
}
```

We ran the above example through CodePal.ai[2], a service which provides commercial coding helpers and development assisting tools. The platform offers automatic C to R conversion and, when fed the above example, produced the following R translation:

[2] https://codepal.ai/

```
source <- "Hello, World!"  # Source string
destination <- character(20)  # Destination string

p <- source  # Pointer to source string
m <- destination  # Pointer to destination string

# Copy the source string to the destination string
while (m <- m + 1) {
  m <- p <- p + 1
}

cat("Source: ", source, "\n")
cat("Destination: ", destination, "\n")
```

It is clear that this translation is trying to mimic the structure and syntax constructs of the original source without understanding the intent. First of all, the resulting `while` loop is syntactically incorrect and will not properly execute (in R `m <- m + 1` cannot be evaluated as boolean). Secondly, if the intent is correctly understood the entire `while` loop becomes irrelevant and can be replaced with a single `destination <- source` statement.

We also tried the original C code with the mtSystems[3] translator, which is based on the C2Eif project from ETH Zurich [8]. This translator converts C to Java code and it produced the following output:

```
package demo;

import static com.mtsystems.coot.String8.s8;

import com.mtsystems.coot.String8;

public class DemoTranslation {
    public static void main(String[] args) {
        String8 source = s8("Hello, World!"); // Source string
        String8 destination = new String8(20); // Destination string

        String8 p = source; // Pointer to source string
        String8 m = destination; // Pointer to destination string

        // Copy the source string to the destination string
        while((m = m.shift(1)).set(-1, (p = p.shift(1)).get(-1)) != 0) {
            ;
        }

        System.out.println("Source: " + source);
        System.out.println("Destination: " + destination);
    }
}
```

[3] https://www.mtsystems.com/

Not only does this produce convoluted code that is difficult to understand, but it also includes proprietary classes, making the resulting translation unportable and generally unusable. A much cleaner and concise Java implementation that captures the source intent should look like this:

```
public class Main {
    public static void main(String[] args) {
        String source = "Hello, World!";
        String destination;

        destination = String.valueOf(source);

        System.out.println("Source: " + source);
        System.out.println("Destination: " + destination);
  }
}
```

### 2.4 Not taking advantage of target's capabilities

This is the other side of the coin of what we covered in Section 2.1. It could often be the case that the target language has capabilities and features, which are specifically well suited for solving the problem that's been originally modelled using the source language. A simple pattern-matching approach may fail to recognise this opportunity and may not take advantage of the additional capabilities available in the target language. One such example is given by Wallis[9]. Speaking specifically about translation to Ada, he emphasises that there is a substantial difference between an naive translation and code that takes full advantage of Ada-specific features. In such cases, the construction of high-quality code would mandate not just translation but a complete re-design.

## 3 Source to source translation via Abstract Syntax Trees

Abstract Syntax Trees (ASTs) are a data structure commonly used in programming language theory to represent the structure of source code written in a specific language. Their key characteristic is that they represent the abstract syntactic structure of the code, without the inclusion of every detail present in the real source code. Here is a simple implementation of Euclid's algorithm for finding the greatest common divisor (GCD) of two integers:

```
while b != 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

A corresponding AST for this code is shown in Figure 1. We can immediately see that the AST representation is abstract (e.g. specific language constructs are replaced by general directives like "branch" and "assign"), complete, and language agnostic — we can take the AST given in Figure 1 and generate concrete implementation in any language that can handle branching and integer variables. Source-to-source translators that use Abstract Syntax Trees (ASTs) typically implement the following workflow:

- **Parsing and AST building**: The first step is to parse the input source code written in the source language. Parsing involves breaking down the code into its constituent elements (tokens) and constructing an AST from these tokens. The AST represents the hierarchical structure of the code and captures its syntactic and structural information.
- **AST Manipulation**: Once the AST is constructed, the source-to-source translator can manipulate the tree to perform various transformations. This can involve adding, removing, or modifying nodes in the tree to achieve the desired translation.
- **Semantic Analysis**: Depending on the source and target languages, the translator may need to perform semantic analysis. This involves checking for type compatibility, resolving references to variables or functions, and ensuring that the translated code maintains the intended behavior.
- **Code Generation**: After the AST has been manipulated and any necessary semantic analysis is complete, the translator generates equivalent code in the target language. This involves traversing the modified AST and emitting code in the target language based on the structure and content of the tree.

AST-based translation has a number of advantages. Because ASTs capture the semantics of the code, they provide an innate capability of encoding information about the meaning of the code. This semantic information is crucial for accurate source-to-source translation and enables AST-based translation systems to go a step farther in comparison to pure basic parsing and text manipulation techniques. We have seen early [10] and more recent examples [11] of translators that leverage AST techniques. Using ASTs brings a number of advantages over simple patter matching. First of all, they offer precise control over code generation since they operate on a structured representation of the code. They can also enforce strong typing and strict adherence to language-specific rules, reducing the likelihood of errors.AST systems can and are often used to optimize code generation, as they produce more efficient and faster code. Code generated using ASTs is not only fast but also predictable, which is important for real-time or resource-constrained applications.

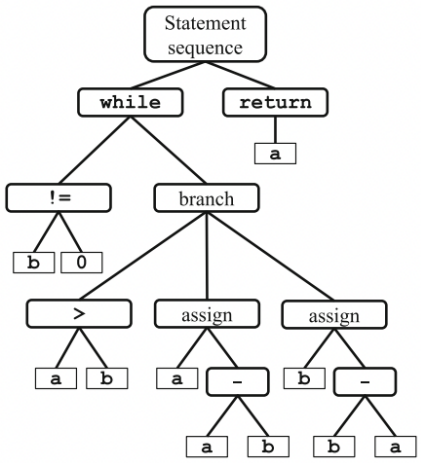


Figure 1: An abstract syntax tree for the Euclidean algorithm as given in Hung-Cuong et al. [12]

On the other hand AST systems are highly language-specific and require significant effort to support new languages or dialects. Developing and maintaining of AST-based translators is complex and time-consuming, as it involves handling the intricacies of different programming languages. Such systems often struggle with unconventional code patterns or languages with non-standard syntax. The biggest drawback of using AST-based translation, however, is the need to have deep skills and understanding of the technology to a level where you can build an efficient translation system. As these systems are heavily language specific and the SAS-to-R translation needs are relatively niche, there is currently no commercially available system that can do such translation out of the box. It goes without saying, that investing the time and effort to build one internally is probably not something that can be commercially justified either.

# 4 Large language models

The start of the transformative evolution of Large Language Models (LLMs) is often attributed to the emergence of the Transformer architecture. This architecture was introduced in the influential "Attention is all you need" paper by Vaswani et al. [13]. Until that time, the state of the art techniques for sequence modelling, which language translation is a major part of, were dominated by recurrent neural networks and LSTMs [14, 15]. This type of time-series models come with a number of limitations that makes them challenging to train. They require complex temporal indexing thus preventing efficient training parallelisation. Historically, they've been also difficult to train due to inherent gradient instabilities [16].

Figure 2 shows the basic Transformer architecture as presented by Vaswani et al. [13]. Transformers are often used in two main variants: the encoder and the decoder. The encoder processes the input sequence, while the decoder generates the output sequence. These stacks consist of multiple layers, each containing the components mentioned above. The key elements on the encoder side of the architecture are as follows:

- **Input Embeddings** — The input sequence of words is typically represented as tokens. Each input token is embedded into a high-dimensional vector space, with the embeddings often learned during training. The purpose of the embeddings is to capture semantic information about the tokens.
- **Positional Encodings** — Since Transformers do not have a built-in notion of word order, positional encodings are added to the input embeddings to provide information about the temporal position of each token in the sequence.
- **Multi-Head Self-Attention Mechanism** — The core of the Transformer is the self-attention mechanism. It allows each token in the input sequence to focus on different parts of the sequence, including itself. The multi-head variant of self-attention allows the model to learn different attention patterns simultaneously.
- **Position-wise Feedforward Networks** — After attention is applied, the output is passed through position-wise feedforward neural networks. These networks are applied independently to each position in the sequence, allowing the model to capture complex, position-specific patterns.

The decoder side of the Transformer architecture is responsible for generating output sequences. Conceptually, the decoder takes the encoded information from the encoder and uses it as context for generating the output sequence. The decoder generates the output sequence one token at a time

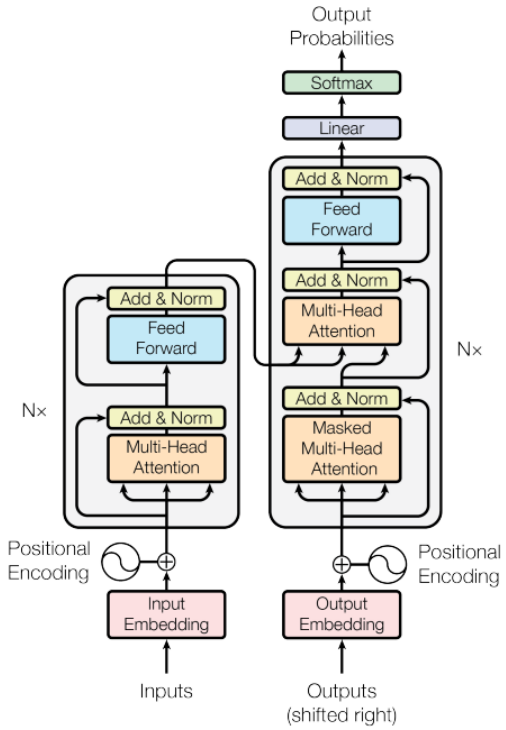


Figure 2: Transformer architecture from Vaswani et al. [13]

in an autoregressive manner — it starts by predicting the first token, and then uses that prediction along with the context to predict the next token.

Transformers have become the backbone of many state-of-the-art language models, such as Bidirectional Encoder Representations from Transformers (BERT) [17] and Generative Pre-trained Transformer (GPT), due to their ability to capture long-range dependencies, parallelize computation, and achieve impressive results in tasks like language translation, text generation, and sentiment analysis. After its introduction in 2022 and in a little over a year, BERT has become "a ubiquitous baseline in Natural Language Processing (NLP) experiments counting over 150 research publications analyzing and improving the model"[18]. As of this writing, Google Scholar returns 6,440 results for scientific articles containing the phrase "Generative Pre-trained Transformer".

## 4.1 Source-to-source translation using LLMs

While LLMs are primarily designed for natural language understanding and generation, they have shown promise in code-related tasks, including source code translation. Understandably, there are certain considerations that make this task challenging for current LLMs:

- **Training data** — Large language models require a substantial amount of training data, but this type of data is very limited when it comes to the source-to-source translation domain. Ideally, we'd like to have access to a large number of code snippets, which implement identical tasks but are written in different programming language.

- **Fine-Tuning** — To make the model more effective for source-to-source translation, fine-tuning on specific code translation tasks may be necessary. This involves training the model on a dataset of source code examples and their corresponding translations. Fine-tuning a model like this, however, may lead to catastrophic forgetting [19]. See Section 5.5 for additional details.
- **Evaluation** — The quality of source-to-source translation should be carefully evaluated, and post-processing may be required to ensure the translated code is syntactically correct and functionally equivalent to the original code. This is more demanding than a simple loss-based evaluation and typically requires a more sophisticated pipeline that can compile, execute, and compare results between the original code and the translated output.

The use of LLMs is also accompanied by challenges around reproducibility, cost control, intellectual property ownership and more, which we cover in details in Section 5. On the other hand, LLMs provide certain advantages over ASTs:

- **High-Level Abstraction** —LLMs can work at a higher level of abstraction because they understand natural language. This makes them more intuitive for developers who may not be well-versed in the details of the programming language.
- **Flexibility** — LLMs can handle a wide range of programming languages and are adaptable to new languages or dialects with minimal effort.
- **No Need for Language-Specific Grammar Rules** — LLMs don't require predefined language-specific grammar rules, making them more scalable across different programming languages.
- **Context Awareness and Intent Understanding** — LLMs can capture context effectively, which helps in understanding and generating code that aligns with the developer's intent.
- **Simpler Deployment** — LLMs can be deployed more easily as they don't require the extensive setup of AST-based systems.

With these advantages in mind, it is clear that LLMs have a substantial potential to solve source-to-source translation tasks. This is partly due to their innate capability to encode knowledge about programming concepts and how they are accomplished using constructs of the language in question. For example, look at the interpretation provided by ChatGPT when asked to explain the toy C problem we showed in Section 2.3.

```
Prompt: What does the following line of C code do: while (*m++ = *p++);

Completion:

The line of C code while (*m++ = *p++); is a concise way to perform a common
    operation in C: copying a null-terminated string (character array) from one
     location to another. Let's break down what it does step by step:
```

```
1. *m++ and *p++: *m++ and *p++ are postfix increment operations. They first
    use the value of m and p, respectively, and then increment the pointers m
    and p. In this context, m and p are assumed to be pointers to characters (
    char*).

2. *m = *p: This part of the code assigns the character pointed to by p to the
    character pointed to by m.

3. while (...): The code is enclosed in a while loop. It continues executing as
     long as the result of the assignment *m = *p is non-zero (true). In C, the
     result of an assignment operation is the value assigned, so as long as
    characters are being copied (i.e., not a null terminator), the loop
    continues.

4. Null termination: The loop will terminate when it encounters a null
    terminator '\0' in the string pointed to by p. This is because the null
    terminator has a value of 0, so when the assignment *m = *p copies the null
     terminator, the loop condition becomes false, and the loop exits.

In summary, this line of code copies the characters from the memory location
    pointed to by p to the memory location pointed to by m until it reaches the
     end of the string (null terminator). It's a concise way to perform string
    copying in C and is commonly used when you want to copy one string into
    another.
```

Here we would rather avoid arguing about the semantics of "understanding", but what is evident from the explanation above is that this LLM model has encoded sufficient knowledge to be able to provide a coherent and correct explanation about the intent of the code. Moreover, unlike the other tools we looked at in Section 2.3, this model can correctly translate the C snippet into R code by focusing on the intent and not so much on the specific syntax. Notice, how the model explicitly provides justification of why it can't perfectly mimic the execution due to the lack of pointer-based operations in R.

```
Prompt: Translate this code to R

Completion:

In R, which is primarily a language for statistical computing and data analysis
    , direct memory manipulation and pointer-based operations like the C code
    you provided are not typically used. Instead, R provides higher-level
    abstractions for working with data.

If you want to copy elements from one vector to another in R, you can use
    standard indexing and assignment operations. Here's an example of how you
```

```
    might copy elements from one vector p to another vector m in R:

m <- character(length(p)) # Create a character vector m of the same length as p
m <- p # Copy the elements from p to m

This code creates a new character vector m of the same length as p and then
    assigns the values from p to m, effectively copying the elements. R takes
    care of the memory management and doesn't require explicit memory
    manipulation like in C.

R is designed for high-level data analysis and manipulation, so it abstracts
    away many low-level memory operations found in languages like C.
```

It also appears that ChatGPT has a sufficient knowledge in SAS and can produce relatively correct translations of SAS code as shown in the following example.

```
Prompt: Translate the following SAS code into R. Produce just the code.

title "Weight Limits on the Y Axis";
proc sgplot data=sashelp.classfit;
  where age > 12;
  band x=name lower=lowermean upper=uppermean;
run;
title;

Completion:

Here is the equivalent R code for the given SAS code:

library(ggplot2)

# Set the title
ggtitle("Weight Limits on the Y Axis")

# Load the data (assuming sashelp.classfit is available in R)
data <- sashelp.classfit

# Filter the data where age > 12
data <- data[data$age > 12, ]

# Create the plot
ggplot(data) +
  geom_ribbon(aes(x = name, ymin = lowermean, ymax = uppermean), fill = "blue")
      +
```

```
labs(x = "Name") +
theme_minimal()
```

Given the suitability of large language models for handling code-related tasks, it is not surprising that their popularity has been on the rise. Earlier this year, Microsoft Chief Executive Satya Nadella announced that to date more than 1 million people had used Microsoft Copilot, an LLM designed to assist with programming tasks [20].

Unfortunately, using LLMs for this type of translation is not as straightforward as it may seem. In the next section we cover some of the key obstacles and discuss how to mitigate some of the challenges that this approach presents.

# 5 Challenges for using LLMs for source-to-source translation

Commercial LLM APIs (also known as "closed-access LLMs") refer to LLMs that are developed and licensed for commercial purposes. These models are typically proprietary, controlled by the organizations or companies that develop them, and are accessible only via APIs for a fee (typically paid per call or per number of tokens). OpenAI's GPT-3, IBM Watson Language Translator, and Microsoft Azure Text Analytics are all examples of such closed-access LLMs. On the other end of the spectrum we have models that are distributed under permissive licenses and can be downloaded and hosted locally (we refer to such models as "open-access LLMs" or "locally hosted", because of the capability to get the entire model and host it in our own environment). Examples for such open-access models are Hugging Face Transformers, BERT, T5 (Text-to-Text Transfer Transformer) and others. In this section we compare and contrast these two categories in the light of their suitability to drive source-to-source translation pipelines.

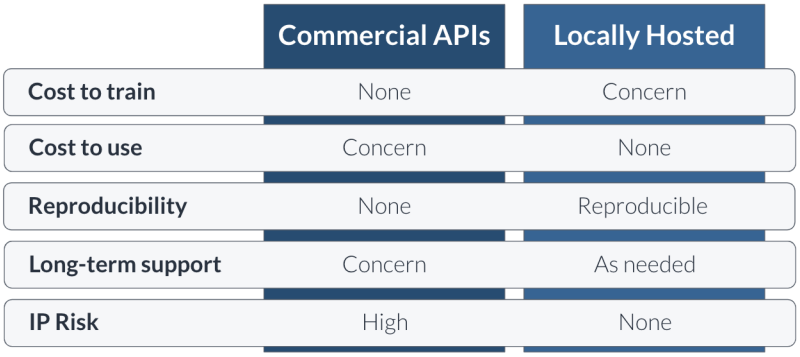


Figure 3: Commercially available LLM APIs versus locally hosted LLMs — advantages and disadvantages.

## 5.1 Lack of reproducibility

It is quite often the case that machine learning models must be reproducible not only in therms of training but also in the outputs they produce. This is especially important in the context of

LLM-based source-to-source translation where the output of the model is code and not artefacts for direct consumption. The resulting code represents an extra layer of complexity between the input data and the expected output and makes traceability even harder.

Reproducible results are key in heavily regulated industries as the pharmaceutical sector and medicine in general. Regulatory bodies require reproducibility as part of compliance and auditing processes. Being able to demonstrate how a model was trained and validated can be essential for legal and ethical reasons. In the health care domain, creating reproducible results is already sufficiently challenging [21] despite existing mandates [22]. The added layer of complexity that source-to-source translation introduces only makes things harder. Moreover, reproducibility is also paramount for addressing critical ethical consideration and for ensuring that the models are fair, unbiased, and free from discriminatory or harmful effects.

In addition, unreproducible outputs of a source-to-source translation pipeline make comparisons between different algorithms, models, and techniques meaningless. At the end of the day, if a source-to-source translator is fed the same original code but outputs different translation for each consecutive execution, how can we tell if the translation model has been improved or not? For example, it has already been shown that ChatGPT exhibits instability between repeated rounds of prompting, suggesting lack of reproducibility [23]. This type of work, which is built on "proprietary models hidden behind opaque API endpoints [...] yields experimental results that are not reproducible and non-deterministic" [24]. In the case of ChatGPT this behaviour has been widely known and is fully expected as this technology never came with a reproducibility promise attached. Moreover, LLMs typically feature purposefully crafted parameters to introduce randomness in their outputs (e.g. `temperature`, `top p` etc. [25]). Such randomness promotes diversity and is often used to fine-tune the trade-off between creativity and coherence in generated text. Although creativity is a desired characteristic in certain domains, when it comes to reproducible translation this is rarely the case.

## 5.2 Loss of IP

A major concern when using external services for source-to-source translation is the increased exposure to intellectual property (IP) risk. Once any information has been sent to a third-party hosted service, this information is no longer subject to the governance and control mechanisms established within the company. In the case of LLMs, such information can be used to retrain the third-party model and diminish the competitive advantage that it may represent. This is a typical "a rising tide lifts all boats" scenario, which although beneficial for the industry, may come at the expense of the individual company. For example, the ChatGPT FAQ clearly states that any information send to the service may be reviewed and used for training [26]. Moreover, OpenAI's Privacy policy [27] reveals that the company collects a wide range of data pertaining to account information, user content, communication, social media, logs, usage statistics, device information, cookies and more. This data is used to "improve our Services and conduct research" and OpenAI "may provide your Personal Information to third parties without further notice to you". In addition, this data is stored in "facilities and servers in the United States and may be disclosed to our service providers and affiliates in other jurisdictions", which introduces additional complications with a number of regulations currently in effect in Europe[4] [28].

[4]ChatGPT was temporarily banned in Italy by the Italian data-protection authority over privacy concerns [29].

## 5.3 Cost control

Training LLMs could be an expensive endeavour. For example, the GPT-3 175 billion parameters model required 3.14E23 FLOPS of computing for training. There are various estimation of how this translates to dollar amount with some analysts suggesting that even at the theoretical 28 TFLOPS maximum of NVIDIA V100 and the lowest 3 year reserved cloud pricing that can be found, executing a single training run for GPT-3 will take 355 GPU-years and cost $4.6M [30]. Others estimate the cost to a whooping $12 million [31]. Undoubtedly if one is using Commercial APIs the cost of training is of no consideration as it is borne by the LLM model provider. There are the "per API call" costs, of course, but they are many orders of magnitude below what a typical LLM training process costs[5] We must not forget that even a locally hosted model still incurs "per call" costs, which are needed for operating the locally hosted hardware. There is a middle ground here where the inference can be moved to the cloud and instances kept shut down when not needed, which makes things really financially efficient. The bottom line, however, is that the majority of the expenses are associated with the training phase. Everyone considering going down the locally trained and hosted model route should consider having a robust cost control measures in place.

## 5.4 Dependency on external technology

It goes without saying that relying on LLMs available via Commercial APIs introduces external dependencies in the translation pipeline. Typically, API providers are incentivized to release new APIs as that's their main vehicle for introducing new functionality and staying competitive. Maintenance of the old APIs, however, is often a burden and just an added cost. For example, if we look at how OpenAI provides access to the GPT models we will see that their models and APIs go through incremental model update cycles. On March 1 they released `gpt-3.5-turbo-0301`. Notice, that the four digit suffix denotes the release date of the model (03/01). The version of the model released on June 13 is designated `gpt-3.5-turbo-0613` and so on. There is also an unnamed version (e.g. `gpt-3.5-turbo`), which points to the latest dated version of the model. OpenAI automatically updates the unnamed version when a newer model becomes available, giving only 2 week notice to their users for the upcoming change. Moreover, after a new model version is launched, older versions are typically deprecated 3 months later [32]. Similarly, the APIs are also being updated with the latest version released on August 22 (`/v1/fine_tuning/jobs`) and the current version (`/v1/fine-tunes`) is set to be shut down on January 04 2024. To their credit, OpenAI are very transparent about such changes and provide comprehensive transition guides. Nevertheless, this certainly introduces overhead, makes reproducibility impossible, and there are no guarantees that other vendors would follow the same transparent approach of sun-setting older model versions and APIs.

## 5.5 Inability to perform fine-tuning

Fine-tuning refers to the process of taking a pre-trained LLM, such as GPT-3 or a similar model, and further training it on a specific task or domain to make it more specialized and useful for that particular application. During fine-tuning, the model is trained on a smaller dataset that is specific to the target task or domain (i.e. source-to-source translation). This dataset contains examples and annotations related to the task, which in our case could be snippets of SAS code with their R equivalents. It has been shown that even foundation models trained on a specific task benefit from fine-tuning on the same task. Li et al. [33] show that StarCoderBase, which was initially trained on

[5] As of the writing of this paper *gpt-3.5-turbo*, the most capable GPT-3.5 model costs about $20 per 10 million tokens (7.5 million words).

The Stack [34], can be further fine-tuned on another 35B Python tokens resulting in a LLM model that substantially outperforms existing LLMs that are also fine-tuned on Python.

Overall, LLM fine-tuning is a powerful technique because it leverages the general language capabilities of the pre-trained model while tailoring it to specific applications. It reduces the need for training large models from scratch, which can be computationally expensive and data-intensive (see Section 5.3). Unfortunately, fine-tuning often leads to a phenomenon known as "catastrophic forgetting" [35]. This refers to an undesired effect where a model forgets or significantly degrades its performance on previously learned tasks when it is further trained on new tasks or data. In essence, the fine-tuning process causes the model to make significant changes to its internal representations. These changes then lead to the model forgetting or erasing previously learned knowledge that were important for earlier tasks.

Generally speaking, closed-access LLMs can't be fine-tuned. Fine-tuning requires access to the foundation model in a way that modifies it and understandably companies that provide commercial LLM APIs don't allow this. Moreover, fine-tuning a model for one specific customer's use-case can inevitably degrade its performance for other tasks and customers. This is not the case with locally hosted open models where we don't really care about catastrophic forgetting as we aren't trying to build a "one size fits all" model. Furthermore, we could fine-tune multiple versions of the same foundation model and run them concurrently (e.g. one for SAS to R translation, one for Python to R translation etc.)

## 5.6 Fair use concerns

Fair use is another concern that must be considered as it could have substantial reputational and financial implications. As already established, LLMs need a large amount of training data in order to learn the relationships needed for generating completions. It is somewhat likely that when they generate code they may reproduce something that was in their original training data. Lemley and Casey [36] suggest that when models generate novel content the use of copyrighted data for their training should be considered fair use. This comes under the provisions around transformative use of the copyrighted source. On the other hand, if a completion is identical or even very similar to something that was in the model's original training data this might not be the case. The situation can be further exacerbated if the completion affects the original creator in a commercial way [37]. This is not a purely theoretical speculation. In November 2022, two developers filed a putative class action, using the pseudonyms J. Doe 1 and J. Doe 2, alleging that Copilot and Codex were trained on the plaintiffs' copyrighted computer code [38]. In September the Authors Guild and 17 authors filed a class-action suit against OpenAI in the Southern District of New York for copyright infringement [39]. The lawsuit claims that OpenAI used their books to train its ChatGPT chatbot. In August the New York Times also stated that they are exploring whether to sue OpenAI to protect the intellectual property rights associated with its reporting [40]. There are generally two ways to de-risk fair use violations:

- **Commitment to protect end-users**: This is similar to how IBM and RedHat reacted during the SCO Group's lawsuit against Linux in the early 2000s [41]. IBM not only vigorously defended itself but also made a strong commitment to protect its Linux users against claims from SCO Group. At the same time Red Hat established the Open Source Now Fund, which aimed to provide legal support to open-source software projects and users facing legal challenges, including those related to SCO's claims [42]. Unfortunately, as to our knowledge none of the major player in the generative AI space have committed to providing similar level of reassurance to their customers.

- **Full control over the training data**: The fair use concerns can be fully mitigated if we are in full control of the training data. If we know that the foundation model has been trained exclusively on data that is in the public domain or has permissive license and if we are in full control of any additional data used for fine-tuning then infringement risks should be of no concern. One good example of permissive dataset for source-to-source translation is The Stack [34], which contains source code data from 358 languages (including SAS) resulting in a near-deduplicated dataset of 6TB in size. This dataset is available under various permissive licenses and can be freely accessed here: `https://huggingface.co/datasets/bigcode/the-stack`. There are already a number of LLMs trained on The Stack like StarCoder and StarCoderBase (15.5B parameters) [33], MosaicML MPT-7B [43], SantaCoder [44] and others.

Overall, while large language models can be a valuable tool for source-to-source translation, they are not a silver bullet, and their effectiveness depends on various factors, including the quality and quantity of training data, fine-tuning, and the specific translation task at hand. Specialized models and approaches may be needed for complex or domain-specific source-to-source translation tasks.

# 6 A blueprint for successful and risk-free source-to-source translation

In order to address the key challenges in a manner that would allow us to build a robust source-to-source translation pipeline, we propose the following architecture. The key design principles underpinning the solution are:


Figure 4: A high-level diagram of a source-to-source translation pipeline, which encapsulates the key elements for a reproducible, flexible, cost efficient, and highly performant solution.

- **Local hosting** — Make sure you have capability to locally train and host the model. Using an open-access, locally hosted model provides the highest level of flexibility. It shields the company from exposing sensitive data, helps it retain its intellectual property, and enables it to tweak the model to its specific needs. None of these benefits are available with closed-access LLMs.

- **Use of foundation models** — Training models from scratch can be very costly. We can significantly accelerate the building of the model and substantially reduce the training costs by starting with a pre-trained foundation model. Ideally, our solution would allow us to bootstrap the model building by providing direct integration with popular repositories that host open-use foundation models (e.g. Hugging Face, PyTorch Hub etc.)

- **Flexible infrastructure** — The solution should provide access to flexible on-demand backend compute for fine-tuning, validating, and hosting the translation pipeline. The compute could be located in the cloud or on-premise. Albeit the flexibility of cloud hosting has a substantial appeal, when it comes to training deep learning models it has been demonstrated that the on-premise compute is often faster and offers significantly better ROI [45]. In an ideal world the solution could provide some kind of a hybrid model, as the recommendation for businesses with heavy GPU workloads is a mix of owned and rented GPUs where guaranteed demand runs on owned GPUs and variable demand runs on the cloud [46].

- **Containerization** — Kubernetes enables efficient scaling, resource management, and orchestration of containers, and can provide a seamless transition between training and deployment and between cloud and on-premise compute. Using containers to facilitate the pipeline is one of our strong recommendations.

- **Reproducibility principle** — Reproducibility is a cornerstone of the scientific method and ensures that tests and experiments can be reproduced by different teams using the same method. In the context of LLMs, reproducibility means that everything needed to recreate the model and its results such as data, tools, libraries, frameworks, programming languages and operating systems, have been captured, so with little effort the identical results are produced regardless of how much time has passed since the original project. Reproducibility is not constrained to model training alone, but should be ensured in the inference pipelines too (i.e. making sure that the translation outputs are always consistent). It also often mandated by regulators (e.g. GxP compliance). This is a really challenging problem as it must capture a wide range of elements and also make sure they are always available (e.g. old versions of packages, data lineage, infrastructure capabilities etc.) Existing open-source frameworks like Docker and MLflow provide baseline mechanisms for implementing reproducibility, portability, and dependency management.

- **Cost control** — Have robust cost control mechanisms in place. The ideal solution should be able to both monitor and enforce cost policies, and should be able to apply such constraints to both storage (via quotas) and back-end compute. The solution should be able to work with both on-premise hardware and also provide integration with the most popular cloud providers (e.g. AWS, GCP, Azure). One solution that is gaining lots of traction in this space is Kubecost[6], which is open-source (Apache 2.0 license) and provides cost allocation, monitoring, optimisation insights, and alerts for Kubernetes spending. Having an efficient cost-control mechanism is also a prerequisite for quantifying the impact of the automated translation.

- **Validation pipeline & degradation detection** — Validation must be a key element of any source-to-source translation pipeline. This could be automated by running the original source, performing the translation, executing the translated code and comparing the outputs. Ensuring that the outputs match within a certain margin of error is considered the gold

[6] `https://www.kubecost.com`

standard for translation validation. This, however, is not always possible. Consider code that produces various plots as its output. The source and target ecosystems would likely use different plotting packages so the results can't be pixel perfect and a data-level comparison becomes meaningless. For such cases we need a human-in-the-loop approach. Being able to execute translations and funnel them through a hybrid validation process puts demands for complex orchestration capabilities on the solution. One solid approach would be for the solution to expose its main functionality via REST APIs thus enabling a complex orchestration framework to drive the validation (e.g. Apache Airflow[7]).

- **Constant feedback loop** Once the translation pipeline is up and running it would need a robust feedback loop in place. This feedback should provide both capabilities to monitor the performance of the translation (in terms of throughput and translation quality) and also implement a mechanism where corrected translations can be fed back to the system and use for future model fine-tuning. This capability where corrections can be fed back and the model improved based on internal institutional knowledge represents the crown jewel of the solution.

[7] https://airflow.apache.org/

# References


[1] Stephen Cass, The Top Programming Languages 2023, IEEE Spectrum, August 2023, https://spectrum.ieee.org/the-top-programming-languages-2023

[2] Frank DiIorio, Jeff Abolafia, *Statistical Programming: Revolution, Evolution, or Status Quo?*, PharmaSUG 2018 - Paper LD-19, https://www.pharmasug.org/proceedings/2018/LD/PharmaSUG-2018-LD19.pdf

[3] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, T. Margaria, *Software Engineering and Formal Methods*, Communications of the ACM, Vol. 51 No. 9, 2008, p. 55. doi:10.1145/1378727.1378742

[4] *MeS-86™ ASSEMBLY LANGUAGE CONVERTER OPERATING INSTRUCTIONS FOR ISIS-II USERS*, Manual Order No. 9800642-0, 1979, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, https://www.nj7p.info/Manuals/PDFs/Intel/9800642B.pdf

[5] Roger Taylor, Phil Lemmons, *Upward migration – Part 1: Translators – Using translation programs to move CP/M-86 programs to CP/M and MS-DOS*, 1982, BYTE, Vol. 7, no. 6. BYTE Publications Inc., pp. 321–344, https://www.tech-insider.org/personal-computers/research/acrobat/8206-b.pdf

[6] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Beatrice Creusillet, François Irigoin, PierreJouvelot, *Pips: a Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers Technical paper*, May 1996, Centre de Recherche en Informatique, Ecole Nationale Superieure des Mines de Paris, 35, rue Saint-Honore, F-77305 Fontainebleau cedex France

[7] SO/IEC 9899:1999(E) – Programming Languages – C, pp. 94, https://www.dii.uchile.cl/~daespino/files/Iso_C_1999_definition.pdf

[8] Marco Trudel, *C nach Eiffel: Automatische Übersetzung und objektorientierte Umstrukturierung von Legacy Quelltext*, 2014, Software Engineering, https://se.inf.ethz.ch/people/trudel/papers/SE2014.pdf

[9] P. J. L. Wallis, *Automatic Language Conversion and Its Place in the Transition to Ada*, Proceedings of the 1985 annual ACM SIGAda international conference on Ada (SIGAda '85), Vol. 5, No. 2, 1985, pp. 275-284.

[10] Paul F. Albrecht, Phillip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg Brückner, *Source-to-source translation: Ada to Pascal and Pascal to Ada*, 1980, In Proceedings of the ACM-SIGPLAN symposium on The ADA programming language (SIGPLAN '80), Association for Computing Machinery, New York, NY, USA, 183–193. https://doi.org/10.1145/800004.807949

[11] G. D. Balogh, G. R. Mudalige, I. Z. Reguly, S. F. Antao and C. Bertolli, *OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling*, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Dallas, TX, USA, 2018, pp. 59-70, doi: 10.1109/LLVM-HPC.2018.8639205.

[12] Nguyen Hung-Cuong, Thang Huynh Quyet, Tru Ba-Vuong, *Rule-Based Techniques Using Abstract Syntax Tree for Code Optimization and Secure Programming in Java*, 2014, 10.1007/978-3-319-14227-2_17

[13] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin, *Attention is all you need.*, Advances in neural information processing systems, 30 (2017).

[14] Ahmed Tealab, *Time series forecasting using artificial neural networks methodologies: A systematic review*, Future Computing and Informatics Journal, Volume 3, Issue 2, 2018, Pages 334-340, ISSN 2314-7288, https://doi.org/10.1016/j.fcij.2018.10.003

[15] Sepp Hochreiter, Jürgen Schmidhuber, *Long Short-Term Memory*m Neural Comput 1997, 9 (8): 1735–1780, https://doi.org/10.1162/neco.1997.9.8.1735

[16] Pascanu, R., Mikolov, T. and Bengio, Y., *On the difficulty of training recurrent neural networks*, May 2013, International conference on machine learning (pp. 1310-1318). Pmlr.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.*, 2019, {https://api.semanticscholar.org/CorpusID:52967399}

[18] Anna Rogers, Olga Kovaleva, Anna Rumshisky, *A Primer in BERTology: What We Know About How BERT Works*, 2020, Transactions of the Association for Computational Linguistics, 8: 842–866, doi:10.1162/tacl_a_00349. S2CID 211532403.

[19] Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, Yuechen Zhang, *An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning*, 2023, ArXiV, abs/2308.08747,https://api.semanticscholar.org/CorpusID:261031244

[20] Jeffrey Dastin, *Microsoft attracting users to its code-writing, generative AI software*, Jan 2023, Reuters, https://www.euronews.com/next/2023/01/25/microsoft-results-ai

[21] Andrew L. Beam, Arjun K. Manrai, Marzyeh Ghassemi, *Challenges to the Reproducibility of Machine Learning Models in Health Care*, JAMA, 2020 Jan 28, 323(4):305-306, doi: 10.1001/jama.2019.20866, PMID: 31904799, PMCID: PMC7335677.

[22] Food and Drug Administration, U.S. Department of Health and Human Services, Center for Drug Evaluation and Research (CDER), *Bioanalytical Method Validation; Guidance for Industry; Availability*, 05/22/2018, 2018-10926, https://www.fda.gov/files/drugs/published/Bioanalytical-Method-Validation-Guidance-for-Industry.pdf

[23] Ben J. Marafino, Vincent X. Liu, *Performance of a large language model (ChatGPT-3.5) for Pooled Cohort Equation estimation of atherosclerotic cardiovascular disease risk*, medRxiv 2023.08.11.23293957, doi: https://doi.org/10.1101/2023.08.11.23293957

[24] Xueguang Ma, Xinyu Crystina Zhang, Ronak Pradeep, Jimmy Lin, *Zero-Shot Listwise Document Reranking with a Large Language Model*, May 2023, ArXiv abs/2305.02156, https://arxiv.org/abs/2305.02156

[25] John Joon Young Chung, Ece Kamar, S. Amershi, *Increasing Diversity While Maintaining Accuracy: Text Data Generation with Large Language Models and Human Interventions*, Annual Meeting of the Association for Computational Linguistics, 2023, https://arxiv.org/abs/2306.04140

[26] *What is ChatGPT? Commonly asked questions about ChatGPT*, OpenAI, https://help.openai.com/en/articles/6783457-what-is-chatgpt

[27] OpenAI OpCo, LLC, *Privacy policy*, June 23, 2023, https://openai.com/policies/privacy-policy

[28] *EU General Data Protection Regulation (GDPR)*, Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), OJ 2016 L 119/1., https://gdpr-info.eu/

[29] Shiona McCallum, *ChatGPT banned in Italy over privacy concerns*, April 2023, BBC, https://www.bbc.co.uk/news/technology-65139406

[30] Chuan Li, *OpenAI's GPT-3 Language Model: A Technical Overview*, 2020, Lambda Blog, https://lambdalabs.com/blog/demystifying-gpt-3

[31] Kyle L Wiggers, *OpenAI launches an API to commercialize its research*, Venture Beat, 2020, https://venturebeat.com/ai/openai-launches-an-api-to-commercialize-its-research/

[32] OpenAI, *Deprecations*, Official documentation, visited 5 Oct 2023, https://platform.openai.com/docs/deprecations

[33] Li, Raymond et al., *StarCoder: may the source be with you!*, 2023, ArXiv, abs/2305.06161, https://api.semanticscholar.org/CorpusID:258588247

[34] Denis Kocetkov and Raymond Li and Loubna Ben Allal and Jia Li and Chenghao Mou and Carlos Muñoz Ferrandis and Yacine Jernite and Margaret Mitchell and Sean Hughes and Thomas Wolf and Dzmitry Bahdanau and Leandro von Werra and Harm de Vries, *The Stack: 3 TB of permissively licensed source code*, 2022, ArXiv, abs/2211.15533, https://api.semanticscholar.org/CorpusID:254044610

[35] Michael McCloskey, Neal J. Cohen, *Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem*, Psychology of Learning and Motivation, Academic Press, Volume 24, 1989, Pages 109-165, ISSN 0079-7421, ISBN 9780125433242, https://doi.org/10.1016/S0079-7421(08)60536-8

[36] Mark A. Lemley, Bryan James Casey, *Fair Learning*, Intellectual Property: Other eJournal, 2020, https://api.semanticscholar.org/CorpusID:219342558

[37] Amanda Levendowski, *How Copyright Law Can Fix Artificial Intelligence's Implicit Bias Problem*, 2018, Georgetown University Law Center, https://scholarship.law.georgetown.edu/facpub/2439/

[38] Skadden, Arps, Slate, Meagher, Flom, *Ruling on Motion To Dismiss Sheds Light on Intellectual Property Issues in Artificial Intelligence*, 2023, JD Supra, https://www.jdsupra.com/legalnews/ruling-on-motion-to-dismiss-sheds-light-6984451/

[39] Alexandra Alter, Elizabeth A. Harris, *Franzen, Grisham and Other Prominent Authors Sue OpenAI*, 2023, The New York Times, Franzen,GrishamandOtherProminentAuthorsSueOpenAI

[40] Bobby Allyn, *New York Times considers legal action against OpenAI as copyright tensions swirl*, 2023, NPR, https://www.npr.org/2023/08/16/1194202562/new-york-times-considers-legal-action-against-openai-as-copyright-tensions-swirl

[41] Stephen Shankland, *SCO sues Big Blue over Unix, Linux*, 2023, CNET, https://www.cnet.com/tech/services-and-software/sco-sues-big-blue-over-unix-linux/

[42] Red Hat, *Red Hat Takes Aim at Infringement Claims*, 2023, Press release, https://www.redhat.com/en/about/press-releases/271

[43] The MosaicML NLP Team, *Introducing MPT-7B: A New Standard for Open-Source, Commercially Usable LLMs*, 2023, https://www.mosaicml.com/blog/mpt-7b

[44] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alexander Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, J. Poirier, Hailey Schoelkopf, Sergey Mikhailovich Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Franz Lappert, Francesco De Toni, Bernardo Garc'ia del R'io, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luisa Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Christopher Hughes, Daniel Fried, Arjun Guha, Harm de Vries, Leandro von Werra, *SantaCoder: don't reach for the stars!*, 2023, ArXiv, abs/2301.03988,https://api.semanticscholar.org/CorpusID:255570209

[45] Chuan Li, *V100 Server On-Prem Vs AWS P3 Instance Cost Comparison*, 2019, Lambda Labs Blog, https://lambdalabs.com/blog/8-v100-server-on-prem-vs-p3-instance-tco-analysis-cost-comparison

[46] Cem Dilmegani, *Cloud GPU: Pricing, Availability, $ / Performance in 2023*, September 2023, AI Multiple Research, https://research.aimultiple.com/cloud-gpu/

[47] OpenAI, *Improving language understanding with unsupervised learning*, June 11, 2018. Archived from the original on 2023-03-18, https://web.archive.org/web/20230318210736/https://openai.com/research/language-unsupervised