

Bridging the Interoperability Gap Using Large Language Models and STITCHES

Javier E. Garza
Lockheed Martin Corporation
Fort Worth, TX
Javier.Garza@lmco.com

Sekinat Y. Quadri
Independent Researcher
Fort Worth, TX
Sekinatt.q@gmail.com

ABSTRACT

The Department of Defense (DoD) is committed to enhancing the interconnectivity of Artificial Intelligence (AI) and platforms for Joint All Domain Command and Control (JADC2), requesting \$1.8 billion for these capabilities in 2024. This is forecast to increase substantially as the budget requests for 2025 and 2026 are being developed, since the DoD is looking for more opportunities for increasing integration between platforms. Traditionally, the DoD's focus on specific platforms has led to development of capable platforms that are incapable of communicating without significant redesign and engineering efforts. Many have tried to solve this interoperability gap by developing global standards that platforms should implement, however, as standards have been developed, they have evolved, leading to the same lack of interoperability to exist between disparate systems. Others, such as the Defense Advanced Research Project Agency (DARPA), have developed technologies enabling rapid integration between systems, one of which is known as the System of Systems Toolchain for the Integration of Heterogeneous Electronic Systems (STITCHES). While this technology is useful for rapid integration, the learning curve for its Domain Specific Language (DSL) can be difficult for those who have no functional programming experience. How can DoD organizations increase their integration velocity while leveraging the latest tools available from DARPA and taking advantage of the latest developments in the field of AI?

One solution is to employ Large Language Models (LLMs) to facilitate the development of STITCHES DSL translation code. In 2022, OpenAI opened many possibilities with ChatGPT. Many open source LLMs have now been developed, some specifically trained to write correct code. This paper will explore the viability of utilizing an open source LLM to effectively construct STITCHES transforms between publicly available standards by developing a dataset based on the DSL, training/fine-tuning it, and finally, explain how our LLM performed on various prompts.

ABOUT THE AUTHORS

Javier Garza is an Associate Fellow and Autonomy, Artificial Intelligence (AI) & Machine Learning (ML) Engineering Program Manager at Lockheed Martin Skunk Works®. Mr. Garza leads the Advanced Sensing AI Portfolio, and his work focuses on architecture, development, integration, testing, and deployment of machine learning algorithms into military platforms. He has more than fourteen years of experience in software engineering, open architecture technologies, flight test, and technical leadership. Mr. Garza has worked on Defense Advanced Research Project Agency (DARPA) programs such as System of Systems Integration, Technology, and Experimentation (SoSITE) and Target Recognition and Adaption in Contested Environments (TRACE). Additionally, he has worked on several Internal Research and Development projects at Lockheed Martin and NASA Mission Control Center (MCC) software modernization projects. Mr. Garza holds a bachelor's degree in Computer Science, a master's degree in Software Engineering from the University of Texas at El Paso, and is currently pursuing a Doctor of Engineering in Artificial Intelligence and Machine Learning at The George Washington University.

Sekinat Quadri is an independent researcher with interests in a variety of fields, including, but not limited to, Artificial Intelligence, Data Analytics, and Healthcare. Previously, she worked in the pharmaceutical and medical fields during the height of the COVID-19 pandemic, and contributed to a published nutrition book. Ms. Quadri holds a bachelor's degree in Biology with a minor in Chemistry and a master's degree in Advanced Data Analytics with a concentration in Health Data Analytics.

Bridging the Interoperability Gap Using Large Language Models and STITCHES

Javier E. Garza
Lockheed Martin Corporation
Fort Worth, TX
Javier.Garza@lmco.com

Sekinat Y. Quadri
Independent Researcher
Fort Worth, TX
Sekinatt.q@gmail.com

INTRODUCTION

Generative Artificial Intelligence (AI), and more specifically, Large Language Models (LLMs), have exploded in popularity since the release of ChatGPT in 2022. Given the rapid developments in this domain, some industries are beginning to lobby for governmental oversight and further protection to protect their intellectual property. One example of this is the New York Times lawsuit against OpenAI and Microsoft, which alleges that ChatGPT leaked its articles word for word (Reed, 2024). Others, such as those in the freelance writing industry, have recently been involved in protests, stating that AI threatens their careers (Gordon, 2024). As an example, Figure 1 provides a drawing of the state-of-the-art capabilities of generative AI, using OpenAI DALL-E, given a specific prompt and its equivalent interpretation of the prompt, drawn manually by one of the authors (Quadri, 2024). While the styles used to develop the image vary between both methods, such technologies can still be useful for obtaining new ideas for tasks such as clean sheet designs of DoD platforms.

The Department of Defense (DoD) recognizes the importance of LLMs as potential boosters of productivity (Harper, 2024). LLMs can be especially useful as engineering aids that could shorten the amount

of time it takes to integrate systems within the DoD, however, utilization of LLMs that are connected to the internet or trained on unknown data introduces a plethora of security vulnerabilities that could undermine national security. Moreover, new models are released often, and it is difficult if not impossible to determine the lineage of the data on which they were trained. Nevertheless, LLMs can be used to generate domain specific data, and they are effective at generating source code for a variety of programming languages.

The DoD has a history of developing systems with fixed configurations that are designed to last for decades. However, this approach makes it difficult to incorporate new technologies and updates. There have been attempts to address this issue by promoting open architectures, but this effort has its own set of challenges. For example, achieving global interoperability through a common architecture is a complex and time-consuming process that may not always result in the most effective solutions. This raises questions about whether there are alternative approaches to achieving interoperability that don't rely on a single, shared interface. The System of Systems Integration Toolchain for Heterogenous Electronic Systems (STITCHES) was developed by Apogee Research through DARPA to address these concerns (Apogee Research, n.d.). At its core, STITCHES uses a functional Domain Specific Language (DSL) to perform translations between different standards. The DSL can be complex and difficult to understand syntactically for traditional programmers. Historically, this has led to two perceptions of STITCHES, one of which is that it can help integrate anything rapidly, and the other is that it is too difficult to wield as a tool.



Figure 1. Generated Image Example (left) and Manual Drawing by S. Quadri (right)

One way to close this gap and ensure that integration remains rapid for DoD engineers and integrators is to employ the usage of a custom LLM that can assist with the generation of STITCHES DSL code.

This paper begins by providing background information on STITCHES and its potential applicability to performing the integration of two publicly available standards, Automatic Dependent Surveillance-Broadcast (ADSB) and Distributed Interactive Simulation (DIS). Then, it describes background literature related to the evolution of LLMs, methods for tailoring them to specific use cases, and similar efforts to perform code generation using them. A description of the approaches and techniques that were attempted is provided, detailing the process ultimately used to develop a dataset for performing fine-tuning on one of the most recent, publicly available, pre-trained language models, Phi-3, which is technically classified as a Small Language Model (SLM). The fine-tuning process is then described. Results and findings after developing prompts of varying complexity and expected code generation snippets in the STITCHES DSL are explored. Finally, the paper provides an overall assessment of the viability of using LLMs for code generation of STITCHES DSL in a DoD context.

BACKGROUND

STITCHES Overview & Applicability

STITCHES enables machine-to-machine communication between systems that weren't initially designed to interoperate. Rather than developing an adapter for each system pairing (Figure 2) if no global standard exists, one can leverage the ability of the toolchain to maintain the representation of a system's interfaces such that it can be transformed into any other system in the ecosystem incrementally (Figure 3). For example, if we consider three messages, *A*, *B*, and *C*, and if no transforms exist from *A* to *C*, but a transform exists between *A* and *B* and another transform exists between *B* and *C*, the toolchain can find a path from *A* to *C*. This is accomplished through a mechanism known as the Field and Transform Graph (FTG). FTGs have two major components, which are Fields and Transforms. A Field can be thought of as the representation of the attributes of a message that can be sent or received by a given system, where its attributes can be defined as other fields or primitive types. Transforms are the mechanism by which fields can be translated into other fields, residing at their destination. In addition to single input transforms, one can leverage multi-input transforms, where more than one input message is required to produce an output, as well as multi-input transforms with synchronization, which allow for more complex logic, such as, for example, triggering the output message to be constructed once a specific input field has been set to a specific value. Transforms must be implemented using the functional DSL developed for STITCHES. Systems that are STITCHES-enabled can be connected to each other via a System of Systems (SoS) specification, and during compilation, transforms based on those connections are rendered automatically into a pre-configured programming language, such as Java or C++.

One example of a use case that can be considered where STITCHES could be useful is a scenario in which a simulation requires live data in real-time from current flights. One of the goals of such simulation would be to ensure that there is deconfliction between the trajectories of the live airplanes and any simulated entities. From a simulation perspective, the DIS standard, could be used. This standard, which is defined by IEEE Std 1278.1-2012, provides definitions of messages, which are known as Protocol Data Units (PDUs). These messages are used to exchange information between simulation services, such as the location of a particular entity, for example. A virtual entity's location is defined as its center in a three-dimensional, right-handed, Cartesian coordinate system. In contrast, ADSB uses the Airborne Position Message to define the location of a platform (Sun, 2021). Within this message, latitude and longitude are defined as 17-bit encoded fields, and altitude is defined as a 12-bit encoded field.

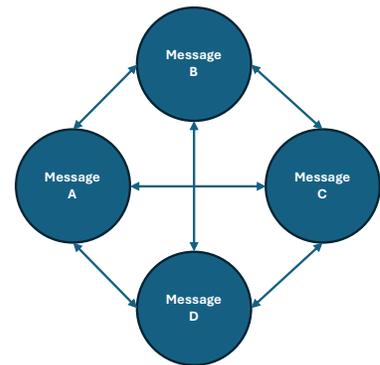


Figure 2. Pairwise Transforms Between All Nodes

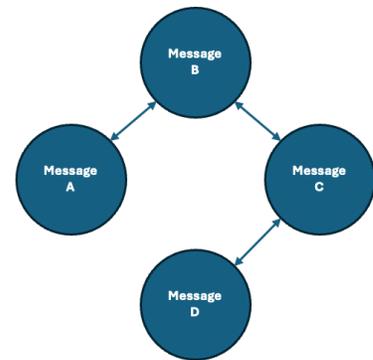


Figure 3. Incremental Transforms Between Messages (STITCHES)

To visualize the real-world platforms in the simulation, it would be necessary to first decode the fields, then translate them to DIS coordinates, all of which could be completed within the STITCHES DSL.

The Evolution of LLMs

Figure 4 shows a general example of the information flow in an LLM. Such models have made significant progress recently, due to breakthroughs in their underlying architectures. More specifically, the Transformer network architecture, proposed by Vaswani et al. (2017), was an important development. The authors found that dominant sequence transduction models showed promise in quality, time, and ability to computationally parallelize. This was significant for interpretability of models with multi-headed self-attention and was particularly noted in the WMT 2014 English-to-German translation task. By inviting variation in transformer components, architectural strengths in performance were able to be noted. In this scenario, basing the transformer solely on attention garnered better results in comparison to recurrent or convolutional layers.

Prior to 2017, standard language models leveraged a unidirectional paradigm. Unidirectional models had limitations, however, models such as Bidirectional Encoder Representations from Transformers (BERT) were designed to be used to pre-train bidirectional representations from unlabeled text (Devlin, Chang, Lee, & Toutanova, 2019). The abilities of BERT are noticeable at token-level tasking as well as sentence-level tasking. By leveraging unsupervised pre-training and fine-tuning, they found that BERT could perform well on many Natural Language Processing (NLP) tasks.

Brown et. al (2020) provided an overview of the process for development of GPT-3, which was trained on 175 billion parameters. The authors described four approaches for in-context learning, of which three were evaluated. The process of fine-tuning, which was not utilized to develop GPT-3, is described as requiring thousands to hundreds of thousands of labeled examples, leading to strong performance. However, one of the disadvantages outlined is the possibility of poor generalization. The other three approaches described are zero-shot, one-shot, and few-shot learning. Zero-shot learning involves only providing the model with a description of a task to follow, and then a prompt. One-shot learning involves providing a description of a task, an example, and a prompt. Few-shot learning expands upon this idea by including more than one example to the model prior to the prompt. The authors found that larger model size was correlated with improved few-shot learning performance.

Limitations exist between LLMs and multimodal data, hence why exploration of Multimodal Large Language Models (MLLMs) for diverse learning is necessary for zero and few-shot multimodal learning. MLLMs differ from traditional LLMs in accordance with their fine-tuning capabilities. MLLMs can perform in context learning from different sources (Huang, et al., 2023). The authors outline KOSMOS-1, which offers an example of a MLLM able to learn in context. The authors found that their MLLM performed significantly well on tasks such as language understanding and generation, perception, and vision tasks.

Thakur et al. (2023) explored the ability of LLMs to support the completion of Verilog code. The authors used existing code from Github and Verilog textbooks to compile datasets to fine-tune pre-trained models. They designed a test suite to evaluate the functional correctness of their fine-tuned models and found that their fine-tuned model showed an increase in performance over GPT 3.5 of 41% when tested with specific problems. Despite this metric, the authors concluded that further improvement could be achieved by utilizing more targeted training data to perform the fine-tuning process or completing more iterations.

In contrast to LLMs which utilize as much data as possible for training, the development of the Phi family of models is based on prior findings that similar performance can be attained by combining filtering of publicly available data

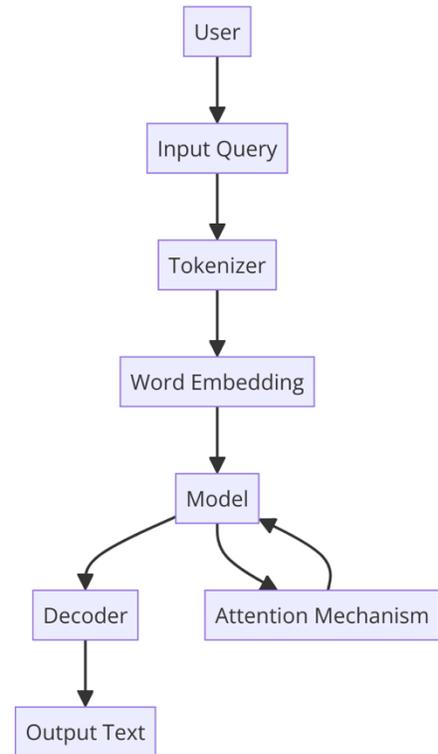


Figure 4. LLM Information Flow

with synthetic data (Microsoft, 2024). The Phi-3 model, for example, was trained on a dataset of only 3.3 trillion tokens, yet according to the authors, has comparable quality to GPT 3.5. By producing a smaller model, it becomes possible to perform inference without requiring the use of specialized or expensive hardware, e.g., on a smartphone. Therefore, this model is an attractive option for DoD use cases, where local inference could be easily integrated onto any air-gapped networks within different security enclaves.

METHODS

Initial Data Collection

While many systems have previously been integrated into the STITCHES ecosystem in various U.S. government-sponsored events, obtaining a publicly releasable, Distribution A release of the toolchain was required to extract a dataset that could be used for experimentation. The FTGs that were distributed with this release provided a set of transforms that could be used to prove out the concepts. Each transform is defined by enclosing “TransformImplementation” XML tags in a CDATA block, as shown in Figure 5. Extraction of all transform DSL was completed by using Python to parse all available FTGs, which are represented as XML files. In total, 324 snippets of DSL with varying complexity were extracted.

In-Context Learning

The most straightforward technique to begin with for this effort was In-Context Learning (ICL), since it required no training. With ICL, a natural language prompt that contains a few examples can be provided as a template to a language model, which uses the prompt and a query as the input to generate the output (Dong, et al., 2024). To determine the viability of ICL, a set of two prompts and answers were developed based on the snippets extracted from the FTGs. Then, a prompt that differed from the original prompt and answer pairs that were provided as examples was provided. This process is shown in Figure 6. The generated code snippet is structured correctly, however, it references “Math.PI”, which is not valid in the DSL. Furthermore, it failed to adequately follow the prompt, since it generated code that multiplies “Time” by pi, instead of X. Although providing more examples to the ICL workflow could yield better results, the number of examples required for the model to generate correct DSL for more than just simple cases could be much larger.

Retrieval Augmented Generation

After attempting ICL, the next logical step was to leverage Retrieval Augmented Generation (RAG) using the FTG specification document provided in the release as a PDF. RAG involves loading text from data, creating small chunks from it, creating numerical embeddings for each chunk, and loading those embeddings into a vector store (LangChain, 2023). A new prompt can then be combined with previous prompts to ensure context is maintained. When the query is sent, relevant documents are retrieved using the embeddings and vector store, and a response

```
<TransformImplementation><![CDATA[
    let pi:double = 3.14;
    let piDeg:double = 180.0;
    Value = input.Value * pi/piDeg;
]]></TransformImplementation>
```

Figure 5. Simple DSL Example

```
Prompt: Create a transform that assigns `Source`,
`Time`, and `Position3D` from an input source.
Answer:
<TransformImplementation><![CDATA[
    Source = input.Source;
    Time = input.Time;
    Position3D = input.Position3D;
]]></TransformImplementation>
Prompt: Create a transform that subtracts 18.0 from the
`Value` subfield of input and sets it to `Value`.
Answer:
<TransformImplementation><![CDATA[
    Value = input.Value - 18.0;
]]></TransformImplementation>".
Prompt: Create a transform that assigns `Source`,
`Time`, and `X` from an input source, and make sure to
subtract 10.0 from time and multiply X by pi.
Answer:
<TransformImplementation><![CDATA[
    Source = input.Source;
    Time = (input.Time - 10.0) * Math.PI;
    X = input.X;
]]></TransformImplementation>
```

Figure 6. In-Context Learning

```
You
Create a transform to convert radians to degrees. Assume your input
source is named "input" and it has a subfield named "Value" of type
double.
< 3 / 3 >

Phi3:Mini
<TransformImplementation><![CDATA[
OutputDegrees = Assign(degrees(Input.Value))
]]></TransformImplementation>

This DSL example directly converts radians to degrees using the built-in
FTG function degrees (.). In this case, "input" is assumed to be a source
field that contains a subfield named "Value" of type double which
represents the value in radians. The expression
degrees(Input.Value) will then convert this value into degrees and
store it in the OutputDegrees variable.

1 FTG_Definition.pdf
```

Figure 7. Prompt Leveraging RAG with FTG Definition

can then be generated by the model. Figure 7 shows an example of a prompt used with RAG with the goal of generating a transform to convert an input from radians to degrees. Initially, one may believe that the syntax produced by the model is correct, however, there are two issues: a missing semicolon and a function call to a non-existent function named “degrees”. The conversion to degrees should either be defined as a function or performed in-line for the code to be usable. In this case, the model was not provided with a strict set of guidelines of how a transform should be structured, and after several different prompts, it was noted that the model performed better at obtaining data verbatim from the provided document than generating the intended code. Given this, a decision was made to next load all available

```
Prompt: Can you show me an example of a transform that converts longitude in radians to longitude in degrees?
Answer:
<Transform>
  <TransformName>transformCommonLonDegToRadLon</TransformName>
  <Source VarName="input">
    <FieldType>[YourOrganization].Common.Lon</FieldType>
  </Source>
  <TransformImplementation><![CDATA[
    // Convert Longitude from Radians to Degrees
    let pi:double = 3.14;
    let piDeg:double = 180.0;
    Value = input.Value * pi / piDeg;
  ]></TransformImplementation>
  <Expense>1</Expense>
</Transform>
```

Figure 8. RAG - Correct Transform Generation Using RAG From All FTGs

FTG XML files through the RAG process, provide a specific template for transform generation as a system prompt, and evaluate the performance of specific prompts. Figure 8 shows an example of where this method performed well, and Figure 9 shows an example of where this method performed poorly. More specifically, the “Source” tag should not have a “SourceType” attribute, there should not be extraneous text below the “FieldType” node, the declarations of the variables are performed incorrectly, and there is

```
Prompt: Can you show me an example of a transform that converts from ECEF to geodetic coordinates?
Answer:
<Transform>
  <TransformName>[ECEFtoGeodeticTransformation]</TransformName>
  <Source VarName="coords" SourceType="Earth-Centered Earth Fixed Coordinate System">
    <FieldType>[YourOrganization.ECEF]</FieldType>
    X_ecef {Value=[Input X coordinate]}
    Y_ecef {Value=[Input Y coordinate]}
    Z_ecef {Value=[Input Z coordinate]}
  </Source>
  <!-- The actual transformation logic will go inside these tags -->
  <TransformImplementation><![CDATA[
    // Example pseudocode for conversion process
    // - Replace with real implementation details
    Decimal latitude = ConvertFromECEFToLatitude(X_ecef, Y_ecef, Z_ecef);
    Decimal longitude = ConvertFromECEFToLongitude(X_ecef, Y_ecef, Z_ecef);
    Decimal
  ]></TransformImplementation>
```

Figure 9. Incorrect Transform Generation Using RAG From All FTGs

an assumption that there is an existing function to convert the Earth-Centered Earth Fixed (ECEF) coordinates to geodetic coordinates. The generated text included an attempt by the model to generate a field definition (not shown), and the transform is incomplete. In summary, this method performs well mostly on tasks that are already pre-defined in the provided, pre-loaded data. One thing that was noted, however, was that this method successfully described transform DSL when prompted with it. The following question was then considered: could it be possible to use RAG to generate the prompts for the DSL of each transform that was extracted from the FTGs, and use them to automatically create a valid dataset that could be used in the fine-tuning process?

Fine-Tuning Dataset Generation & Testing

A Python script was developed to generate each explanation and associate it with its respective transform, outputting it into a JSON file. Within the generated dataset, the explanations for each transform were overly verbose in many cases, which was indicative that this might not be the best approach. The transforms were composed of varying complexity. Any transforms with more complex code yielded lengthy explanations, some of which repeated the code again. Regardless, evaluation of the dataset was necessary. A pipeline that uses Supervised Fine-Tuning Trainer (SFTT) was implemented in Python to perform training. The dataset was split into 80% training and 20% evaluation subsets, and training was performed. The training configuration used a batch size of seven, a learning rate of 0.0001, with gradient accumulation steps of four, and a max sequence length of 128. While the training loss decreased during training over several epochs, the validation loss remained high, as can be seen in Figure 10, This indicated that the labels that were automatically generated for the DSL code

Step	Training Loss	Validation Loss
10	1.628400	1.194418
20	1.145200	1.083789
30	1.021200	1.023164
40	1.022400	0.976231
50	0.965200	0.950904
60	0.915500	0.933253
70	0.935800	0.910253
80	0.878900	0.884461
90	0.859900	0.879399
100	0.646400	0.935072

Figure 10. SFTT Training Metrics with Automatically Generated Labels

snippets were inadequately representing them. Due to this, training was terminated after several epochs to re-evaluate the dataset and fine-tuning method.

At this point, revisiting the originally extracted transforms to select a subset of them for manual labeling was necessary. Transforms were selected based on increasing complexity and ability to be described in a concise statement, and a total of 42 transforms were selected. Manual prompt labels were created for each snippet of DSL code, as shown in Figure 11. The dataset was then augmented to 1,680 total prompts and their expected solutions. This provided enough data for the model to learn its underlying patterns and achieved a balanced distribution of the different classes of transforms that were manually labeled. Once this dataset was created, the fine-tuning pipeline was revised to use Low Rank Adaptation (LoRA). This technique allows for indirect training of some of the dense layers in a neural network by performing optimization on the rank decomposition matrices and maintaining the frozen state of the pre-trained weights (Hu, et al., 2021).

```
"instruction": "Create a transform that assigns `Source`, `Time`, and `Position3D` from an input source.",
"output": "<TransformImplementation><![CDATA[\n    Source = input.Source;\n    Time = input.Time;\n    Position3D = input.Position3D;\n  ]]></TransformImplementation>",
"input": "input",
"prompt": "Below is an instruction that describes a task. Write a response that appropriately completes the request.
### Instruction: Create a transform that assigns `Source`, `Time`, and `Position3D` from an input source.
### Input: input
### Output <TransformImplementation><![CDATA[\n    Source = input.Source;\n    Time = input.Time;\n    Position3D = input.Position3D;\n  ]]></TransformImplementation>"
```

Figure 11. Sample of a Manually Labeled Transform Snippet

Training Configuration

The hyperparameter configuration used for the LoRA method differed slightly from the SFTT method that was previously outlined. The batch size was set as eight, with four gradient accumulation steps, a learning rate of 0.0001, and ten epochs. The LoRA implementation required a definition of several other parameters, such as the attention dimension, the alpha parameter for scaling, and the dropout probability for layers. The attention dimension and alpha parameter were set to 16, and the dropout probability was set to 0.05. The dataset was split into 95% training and 5% validation set for training, resulting in 1596 transforms for training and 84 transforms for validation. Training metrics were tracked using MLflow as shown in Figure 12, and training was completed on an NVIDIA RTX 3090 Ti GPU in less than one hour.

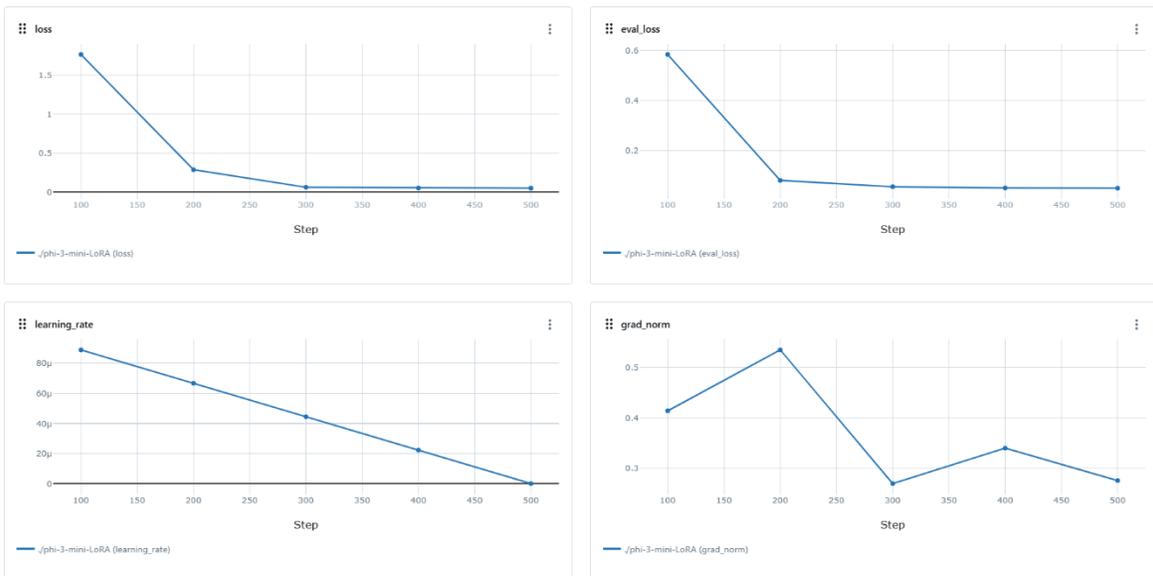


Figure 12. MLflow Training Metrics

RESULTS

As shown, the training metrics indicated that the change in the approach to the definition of the prompts and the augmentation of the data were effective. After training, the output of the LoRA process was merged with the original model. Additional metrics were then calculated to determine whether the model retained the parameters attained during fine-tuning. More specifically, Recall Oriented Understudy for Gisting Evaluation (ROUGE) metrics compare generated text with reference text, producing a similarity score (Amanat, 2024). There are four metrics that were calculated, Rouge-1, which measures the overlap of a single word with the reference text, Rouge-2, which measures the overlap of pairs of words with the reference text, Rouge-L, which measures the overlap of the longest common sequence between the generated text and the reference text, and the Rouge-L sum, which measures the longest common sequence by interpreting new lines as sentence boundaries. Higher scores indicate higher similarity. Figure 13 shows the calculated metrics. With the model successfully fine-tuned, an evaluation of its code generation capabilities could be completed. For initial inference testing, the model temperature was set to 0.3, and a max of 256 tokens was specified.

```
Rouge 1 Mean: 86.77820289410478
Rouge 2 Mean: 81.4781789615524
Rouge L Mean: 85.22682212391098
Rouge Lsum Mean: 86.69764435489962
```

Figure 13. ROUGE Metrics

Prompts of varying difficulty were created and used with the final model. In this section, several of the prompts and their results are outlined. Prompt #1, shown in Figure 15, illustrates a very simple example, where only one field is set using a resolving assign.

```
Prompt: Write a transform that
Assigns Potato into Tomato from
an input source.
Generated (Correct):
<TransformImplementation>
<![CDATA[
    Tomato = Assign(input.Potato);
]]></TransformImplementation>
```

Figure 15. Test Prompt #1

Prompt #2, shown in Figure 17, adds another field, and directly assigns the values, as expected. In contrast, Prompt #3, shown in Figure 14 generated an incorrect transform due to the lack of existence of a function named “uint8cast”, and

```
Prompt: Write a transform that
assigns two fields, 'Area' and
'Volume' from an input source.
Generated (Correct):
<TransformImplementation>
<![CDATA[
    Area = input.Area;
    Volume = input.Volume;
]]></TransformImplementation>
```

Figure 17. Test Prompt #2

Prompt #4, shown in Figure 16 generated a mathematically incorrect transform with incorrect syntax when using number values. At first look, these results seem unfavorable when prompts become more complex. However, the generated code from all the prompts attempted maintained the correct structure of the DSL, which was not always the case for the ICL or RAG approaches. Considering the scenario outlined previously, where one of the goals may be to translate ADSB Position into a DIS Entity State PDU, a prompt was devised to attempt to generate the STITCHES transform for it. Figure 18 shows the results of this prompt. The transform that was generated is syntactically correct.

```
Prompt: Create a transform that
converts a 'Time' subfield of
input to bytes.
Generated (Incorrect):
<TransformImplementation>
<![CDATA[
    byteTime = uint8cast(input.Time);
]]></TransformImplementation>
Correct:
<TransformImplementation>
<![CDATA[
    byteTime = ToBytes(input.Time);
]]></TransformImplementation>
```

Figure 14. Test Prompt #3

```
Prompt: Write a transform that
sets Latitude, Longitude, and
Altitude after converting from
radians to degrees on the input
source, and from feet to meters.
Generated (Incorrect):
<TransformImplementation>
<![CDATA[
    Latitude = 3.1415926535897932385d * \
    (input.Latitude / 3.14159265358979323852d);
    Longitude = 3.1415926535897932385d * \
    (input.Longitude / 3.14159265358979323852d);
    Altitude = 0.3048d * (input.Altitude / 16.0f);
]]></TransformImplementation>
Correct:
<TransformImplementation>
<![CDATA[
    Latitude = input.Latitude * 180.0 / 3.14159;
    Longitude = input.Longitude * 180.0 / 3.14159;
    Altitude = 0.3048 * input.Altitude;
]]></TransformImplementation>
```

Figure 16. Test Prompt #4

Moreover, none of the training data, test data, or validation data that was provided to the model during the fine-tuning process includes any information about ADSB or DIS. This indicates that the original model may have

contained those standards in its training set. Mathematically, the inner calculations of this transform have not been verified.

```
Prompt: Write a transform that converts the geographic coordinates
of an ADSB position message, which are `latitude`, `longitude`,
`altitude` to the `X`, `Y` and `Z` vectors of a Distributed
Interactive Simulation EntityStatePDU message. Do the math for the
conversions.
Generated (Correct):
<TransformImplementation>
<![CDATA[
  // Assigning from source and converting to radians:
  X = 3958.049 * Position.Latitude * 180.0 / 3.141592653589793;
  Y = 3958.049 * Position.Longitude * 180.0 / 3.141592653589793 /
    cos(3958.049 * Position.Latitude * 180.0 / 3.141592653589793);
  Z = Position.Altitude / 100.0;
]]></TransformImplementation>
```

Figure 18. ADSB Position to DIS Position Vector Transform

CONCLUSION AND FUTURE WORK

LLMs are here to stay, as has been proven by the almost-daily advancements that have occurred recently, such as the release of NVIDIA's Nemotron-4, 340B model in June 2024, which can generate synthetic data for training LLMs and SLMs (NVIDIA, 2024). While the DoD has recognized their importance and has begun to find ways to work with them, more work must be completed to ensure that they are applied to the problems that make the most sense. Integration of DoD platforms can be expensive and may require special domain knowledge to be completed adequately. However, it is critical that as models are evaluated and integrated into government networks, the lineage of the data they were trained on should be well understood. Investments into cutting-edge GPUs and MLOps infrastructure must continue to be made by the DoD to ensure that it can rapidly benefit from new innovations. By embracing LLMs and utilizing them to provide ideas for improving internal processes, enhance technical learning, and effectively speed up productivity, many benefits can be realized by the DoD and industry.

This paper demonstrated the feasibility of leveraging LLMs to generate code for the STITCHES DSL. A high-level overview of STITCHES and its applicability was provided. Then, a use case for integrating ADSB and DIS was explored, as well as the evolution of LLMs over the past several years. Three distinct methods were evaluated and detailed, including In-Context Learning (ICL), Retrieval Augmented Generation (RAG), and fine-tuning using Supervised Fine-Tuning Trainer (SFTT) and Low Rank Adaptation (LoRA). Different approaches to generate a usable dataset were explored. After performing fine-tuning on a pre-trained SLM, Phi-3, using the dataset that was created, it was found that in many cases, the expected code snippets were generated from the test prompts of varying complexity. The findings of this work suggest that LLMs and SLMs can be viable tools for generating code in a domain-specific environment such as a DoD context.

One necessary limitation in this work was the utilization of the Distribution A FTG, which contains minimal transform examples compared to the number of transforms available internally to the DoD and contractors. One of the ways in which the dataset could be more representative of the constructs available within the DSL would be to develop more manually labeled data in the future, based on more DSL transform examples. This would likely help resolve the remaining syntax issues and may lead to a DSL code generation capability that is on par with other programming languages that LLMs and SLMs are already trained to generate code for. Finally, after performing fine-tuning, a future task would be to integrate the model into Ollama, which is a framework that serves LLMs locally, and OpenWebUI, which is a front-end user interface, to make the fine-tuned model more usable to others.

REFERENCES

- Amanat, M. U. (2024, January 18). *LLM evaluation with Rouge*. Retrieved from Medium: <https://medium.com/@MUmarAmanat/llm-evaluation-with-rouge-0ebf6cf2aed4>
- Apogee Research. (n.d.). *STITCHES*. Retrieved from Apogee Research: <https://apogee-research.com/stitches/>
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., . . . Ra. (2020). Language Models are Few-Shot Learners. *Advances in neural information processing systems*, 1877-1901.
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019, May 24). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Retrieved from arXiv: <https://arxiv.org/pdf/1810.04805>
- Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., . . . Sui, Z. (2024, June 18). *A Survey on In-Context Learning*. Retrieved from arXiv: <https://arxiv.org/abs/2301.00234>
- Gordon, A. (2024, May 13). *Why Protesters Around the World Are Demanding a Pause on AI Development*. Retrieved from Time: <https://time.com/6977680/ai-protests-international/>
- Grewal, S. (2021, March 9). *Federal Data Management at the Tactical Edge*. Retrieved from Cohesity: <https://www.cohesity.com/blogs/federal-data-management-at-the-tactical-edge/>
- Harper, J. (2024, May 9). *Army set to issue new policy guidance on use of large language models*. Retrieved from DefenseScoop: <https://defensescoop.com/2024/05/09/army-policy-guidance-use-large-language-models-llm/>
- Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., . . . Chen, W. (2021, October 16). *LoRA: Low-Rank Adaptation of Large Language Models*. Retrieved from arXiv: <https://arxiv.org/pdf/2106.09685>
- Huang, S., Dong, L., Wang, W., Hao, Y., Singhal, S., Ma, S., . . . Wei, F. (2023, March 1). *Language Is Not All You Need: Aligning Perception with Language Models*. Retrieved from arXiv: <https://arxiv.org/pdf/2302.14045>
- IEEE. (2012). IEEE Standard for Distributed Interactive Simulation--Application Protocols. *IEEE Std 1278.2-2012 (Revision of IEEE Std 1278.1-1995)*, 1-747.
- LangChain. (2023, February 5). *Tutorial: ChatGPT Over Your Daa*. Retrieved from LangChain: <https://blog.langchain.dev/tutorial-chatgpt-over-your-data/>
- Microsoft. (2024, May 23). *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*. Retrieved from arXiv: <https://arxiv.org/pdf/2404.14219>
- NVIDIA. (2024, June 14). *Nemotron-4 340B Technical Report*. Retrieved from NVIDIA: https://research.nvidia.com/publication/2024-06_nemotron-4-340b
- Quadri, S. (2024). *Mopey Dog with Watermelon Donut Collar*.
- Reed, R. (2024, March 22). *ChatNYT*. Retrieved from Harvard Law Today: <https://hls.harvard.edu/today/does-chatgpt-violate-new-york-times-copyrights/#:~:text=In%20its%20suit%2C%20the%20Times,by%20Wirecutter%2C%20an%20affiliate%20site>
- Sun, J. (2021). *The 1090 megahertz riddle: a guide to decoding mode S and ADS-B signals*. TU Delft OPEN Publishing.
- Thakur, S., Baleegh, A., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R., & Garg, S. (2023). Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention is All You Need. *Advances in neural information processing systems*.