ETSI GS ENI 030 V4.1.1 (2024-03)

Experiential Networked Intelligence (ENI); Transformer Architecture for Policy Translation

Disclaimer

The present document has been produced and approved by the Experiential Networked Intelligence (ENI) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
DGS/ENI-0030v411_Trans_Arch

Keywords
artificial intelligence, cognition, language,
policy management

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights 5
Foreword 5
Modal verbs terminology 5
Executive summary 5
1 Scope 6
2 References 6
2.1 Normative references 6
2.2 Informative references 6
3 Definition of terms, symbols and abbreviations 8
3.1 Terms 8
3.2 Symbols 10
3.3 Abbreviations 10
4 Natural Language Processing with AI 11
4.1 Introduction (informative) 11
4.2 Problems with Natural Language Processing (informative) 11
4.3 Background and Previous Work (informative) 11
4.3.1 Bag of Words Model 11
4.3.2 n-Gram Model 12
4.3.3 Recurrent Neural Networks Model 12
4.3.4 Convolutional Neural Networks Model 14
4.3.5 Long Short Term Memory Model 14
4.3.6 Gated Recurrent Units 16
4.3.7 Attention 17
4.3.7.1 Motivation 17
4.3.7.2 Definition 18
4.3.7.3 Example 18
4.4 Transformers and Large Language Models 19
4.4.1 Motivation (informative) 19
4.4.2 Basic Transformer Architecture (informative) 19
4.4.2.1 Architectural Overview 19
4.4.3 Basic Transformer Models (informative) 21
4.4.3.1 Introduction 21
4.4.3.2 The Original Transformer Architecture 21
4.4.3.3 Transformer Advantages 22
4.4.3.4 Transformer Limitations 22
4.4.4 Other Transformer Models (informative) 23
4.4.4.1 Introduction 23
4.4.4.2 Transformer-XL 23
4.4.4.3 Compressive Transformers 23
4.4.4.4 BERT Models (informative) 24
4.4.4.4.1 The Original BERT Model 24
4.4.4.4.2 RoBERTa 25
4.4.4.4.3 ALBERT 25
4.4.4.4.4 CodeBERT 26
4.4.4.4.5 Additional Optimization Features 26
4.4.4.5 GPT Models (informative) 26
4.4.4.5.1 Introduction 26
4.4.4.5.2 OpenAI GPT Models 26
4.4.4.5.3 BLOOM 28
4.4.4.5.4 PaLM Models 28
4.4.4.5.5 LLaMA Models 29
4.4.4.6 Sparse Transformers (informative) 29
4.4.4.7 T5 Models (informative) 30
4.4.4.8 Mixture of Experts Model (informative) 30

4.4.5 Conclusions (normative) 31
4.5 Prompting for LLMs, Transformers, and Chatbots 31
4.5.1 Introduction (normative) 31
4.5.2 Prompting 31
4.5.2.1 Introduction (normative) 31
4.5.2.2 Input-Output Prompts 32
4.5.2.2.1 Introduction (informative) 32
4.5.2.2.2 Zero-Shot Prompts (informative) 32
4.5.2.2.3 Few-Shot Prompts (informative) 32
4.5.2.3 Self-Consistency Prompts (normative) 32
4.5.2.4 Chain-of-Thought Prompts (normative) 32
4.5.2.5 Tree-of-Thought Prompts (normative) 33
4.5.2.6 Skills-in-Context Prompts (informative) 33
4.5.2.7 Graph-of-Thought Prompts (normative) 34
4.5.2.8 Common Procedures to Increase the Value of Prompting 34
4.5.2.8.1 Introduction (informative) 34
4.5.2.8.2 Active Learning (normative) 34
4.5.2.8.3 Information Retrieval (informative) 35
4.5.3 Prompt Tuning (informative) 35
4.5.4 Prompt Templates (normative) 35
4.5.5 Prompt Pipelines (informative) 37
4.5.6 Prompt Drift (informative) 37
4.5.7 The Use of Prompts in an ENI System (normative) 37
4.6 Instruction Tuning (normative) 39
4.7 The Use of Knowledge Graphs to Improve Reasoning (normative) 39
4.8 Retrieval Augmented Generation (normative) 40

5 Transformer Architectural Requirements (normative) 41
5.1 Introduction 41
5.2 Transformer and LLM Usage in an ENI System 41
5.3 Transformer and LLM Architectural Requirements 42
5.4 Transformer and LLM Reference Point Requirements 42

6 Transformer Architecture for ENI (normative) 43
6.1 Introduction 43
6.2 Design Principles 43
6.2.1 Use Case 43
6.2.2 Overview 43
6.2.3 Using Transformers vs. Traditional Compilers and Parsers 44
6.3 Transformer Management Functional Block Architecture 44
6.3.1 Introduction 44
6.3.2 RAG Framework Functional Block 45
6.3.3 Prompting Framework Functional Block 46
6.3.4 Transformer Processing Functional Block 47
6.3.4.1 Overview 47
6.3.4.2 Choice of Transformer to Use 47
6.3.4.3 Open Source Components Availability for the Transformer Processing FB 48
6.3.5 Output Generation Functional Block 49
6.4 Interaction with Other ENI System Functional Blocks 50

7 Transformer Internal Reference Points (normative) 50
7.1 Introduction 50
7.2 External Reference Points 50
7.3 Internal Reference Points 50

8 Areas of Future Study (informative) 51
8.1 Open Issues for the present document 51
8.2 Issues for Future Study 51

History 52

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Experiential Networked Intelligence (ENI).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

The present document specifies how a transformer architecture may be used to translate intent policies from an end-user, application, or other external source to ENI Policies for use in cognitive networking and decision making in modern system design. The primary use cases are twofold:

1) to translate a natural language intent policy into an ENI Policy that uses a Domain-Specific Language; and

2) to translate an ENI Domain Specific Language intent policy into an implementation that uses a programming language, such as Python™ or Java®.

1 Scope

The present document enhances the information described in ETSI GR ENI 018 [i.1] and specifies how a transformer architecture can be used to translate input policies from an end-user, application, or other external source to ENI Policies for use in cognitive networking and decision making in modern system design.

The present document specifies a transformer-based architecture that can be used to parse, understand, and translate text. This enables different types of input policies to be translated into an appropriate ENI Policy using a transformer architecture.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI GS ENI 005 (V3.1.1): "Experiential Networked Intelligence (ENI); System Architecture".

[2] ETSI GS ENI 019 (V3.1.1): "Experiential Networked Intelligence (ENI); Representing, Inferring, and Proving Knowledge in ENI".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GR ENI 018 (V2.1.1) (08-2021): "Experiential Networked Interlligence (ENI); Introduction to Artificial Intelligence Mechanisms for Modular Systems".

[i.2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, I. Polosukhin: "Attention Is All You Need", 31st Conference on Neural Information Processing Systems, 2017.

[i.3] J. Wei, et al.: "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models", January 2023.

[i.4] Z. Dai, et al.: "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context", 57th Conference for the Association for Computational Linguistics, 2019.

[i.5] J. Rae, et al.: "Compressive Transformers for Long-Range Sequence Modelling", November 2019.

[i.6] J. Devlin, et al.: "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", Google AI, March 2019.

[i.7] Y. Liu, et al.: "RoBERTa: A Robustly Optimized BERT Pretraining Approach", University of Washington and Facebook AI, 2019.

[i.8] Z. Lan, et al.: "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations", International Conference on Learning Representation, February 2020.

[i.9] A. Radford, et al.: "Improving language understanding by generative pre-training", Technical Report, OpenAI blog, 2018.

[i.10] A. Radford, et al.: "Language models are unsupervised multitask learners", OpenAI blog, 1(8):9, 2019.

[i.11] T. Brown, et al.: "Language models are few-shot learners", Advances in Neural Information Processing Systems, 2020.

[i.12] L. Ouyang, et al.: "Training language models to follow instructions with human feedback", Advances in Neural Information Processing Systems, 2022.

[i.13] N. Stiennon, et al.: "Learning to Summarize from Human Feedback", Proceedings of the 34th International Conference on Neural Information Processing Systems, 2020.

[i.14] OpenAI: "GPT-4 Technical Report", March 2023.

[i.15] Big Science Workshop: "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model", June 2023 (final version, v4).

[i.16] A. Chowdery, et al.: "PaLM: Scaling Language Modelling with Pathways", October 2022.

[i.17] P. Barham, et al.: "Pathways: Asynchronous Distributed Dataflow for ML", March 2022.

[i.18] D. Driess, et al.: "PaLM-E: An Embodied Multimodal Language Model", March 2023.

[i.19] Google®: "PaLM 2 Technical Report", May 2023.

[i.20] K. Singhal, et al.: "Large language models encode clinical knowledge", July 2023.

[i.21] H. Touvron, et al.: "LLaMA: Open and Efficient Foundation Language Models", February 2023.

[i.22] H. Touvron, et al.: "Llama 2: Open Foundation and Fine-Tuned Chat Models", July 2023.

[i.23] H. Shirzad, et al.: "Exphormer: Sparse Transformers for Graphs", July 2023.

[i.24] C. Raffel, et. al.: "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", Journal of Machine Learning Research, June 2020.

[i.25] H. Chung, et al.: "Scaling Instruction-Finetuned Language Models", December 2022.

[i.26] S. Zuo, et al.: "MoEBERT: from BERT to Mixture-of-Experts via Importance-Guided Adaptation", April 2022.

[i.27] S. Yao, et al.: "Tree of Thoughts: Deliberate Problem Solving with Large Language Models", May 2023.

[i.28] J. Chen, et al.: "Skills-in-Context Prompting: Unlocking Compositionality in Large Language Models", August 2023.

[i.29] J. Johnson, M. Douze, H. Jégou: "Billion-scale similarity search with GPUs", February 2017.

[i.30] P. Christiano, et al.: "Deep reinforcement learning from human preferences", originally published June 2017, revised February 2023.

[i.31] Z. Feng, et al.: "A Pre-Trained Model for Programming and Natural Languages", September 2020.

[i.32] Y. Wang, et al.: "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation", September 2021.

[i.33] Y. Wang, et al.: "CodeT5+: Open Code Large Language Models for Code Understanding and Generation", May 2023.

[i.34] S. Mahdavi: "Large Language Models Encode Clinical Knowledge", December 2022.

[i.35] P. Lewis, et al.: "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", April 2021.

[i.36] S. Pan, et al: "Unifying Large Language Models and Knowledge Graphs: A Roadmap", June 2023.

[i.37] Y. Wu, et al.: "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", 2016.

[i.38] Microsoft®: "Megatron-DeepSpeed".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

active learning: learning algorithm that can query a user interactively to label data with the desired outputs

NOTE: The algorithm proactively selects the subset of examples to be labelled next from the pool of unlabelled data. The idea is that an ML algorithm could potentially reach a higher level of accuracy while using a smaller number of training labels if it were allowed to choose the data it wants to learn from.

attention: part of a neural architecture that dynamically computes a weighted distribution on input text, assigning higher values to more relevant elements

NOTE: Attention mimics cognitive attention as performed in the human brain. It selectively enhances some parts of the input data while diminishing other parts. Instead of encoding the input sequence into a single fixed context vector, the attention model develops a context vector that is filtered specifically for each output time step. The model then predicts next word based on context vectors associated with these source positions and all the previous generated target words.

batch learning: type of offline learning algorithm that is updated (i.e. retrained) periodically

catastrophic forgetting: tendency of an artificial neural network to forget previously learned information when learning new information

chatbot: computer program that simulates human conversation through text or voice interactions

concept drift: underlying statistical properties of the data an algorithm is trained on change over time

NOTE: Concept drift means that the LLM or transformer is not taking changing data and its meanings into account during inference time.

domain specific language: small human-understandable language that uses a higher level of abstraction to communicate and configure software systems for a particular application domain

embedding: vector that represents the meaning of a word or phrase to the LLM

ensemble model: technique created by combining the predictions of multiple base models

Extended Backus-Naur Fom: notation used to define the syntax of a formal language

foundation model: type of LLM trained on a vast quantity of data at scale (often by self-supervised learning or semi-supervised learning) such that it can be adapted to a wide range of downstream tasks

generative artificial intelligence: type of artificial intelligence that can create new content (e.g. text, images or music) by learning the patterns and structures of existing data and then using those patterns to generate new data that is similar to the original data

graph transformer: type of transformer that generalizes the transformer architecture to graphs

instruction tuning: making a language model more generic by training the model on a large set of varied instructions

language model: use of probabilistic and/or statistical mechanisms to determine the probability of a given sequence of words occurring in a sentence

large language model: type of self-supervised language model whose number of parameters in the model can change autonomously as it learns

mixture of experts model: technique where multiple expert models are used to divide a problem space into homogeneous regions, where only one or a few expert models will be run

one-cold vector: 1 × N matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary, where the vector consists of 1s in all cells with the exception of a single 0 in a cell used uniquely to identify the word

one-hot vector: 1 × N matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary, where the vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word

pipeline: end-to-end construct that orchestrates a flow of events and data in response to a trigger

prompt: input that an LLM responds to

prompt drift: degradation of a model's performance due to changes in the prompts that it is given

prompt engineering: process of designing a set of prompts to generate a specific output

prompt injection: attack against applications that have been built on top of AI models

prompt tuning: efficient, low-cost method of adapting an AI foundation model to new downstream tasks without retraining the model and updating its weights

prompt pipeline: pipeline that takes a user request and translates into a prompt or set of prompts

NOTE: A prompt pipeline typically uses one or more prompt templates and may also use external knowledge.

prompt template: pre-defined structure that can be used to generate text

prompting, chain-of-thought: concatenating a series of prompts that serve as intermediate steps to achieve desired behaviour

prompting, few-shot: prompting an LLM with a small number of examples of expected behaviour

prompting, skills-in-context: instructs an LLM how to compose basic skills to resolve more complex problems

prompting, tree-of-thought: prompting an LLM with a starting point and then asking it to explore different possible outcomes or conclusions that serve as intermediate steps toward problem solving

prompting, zero-shot: prompting an LLM without any examples of expected behaviour

retrieval augmented generation: framework for improving the performance of LLMs by providing them with access to external knowledge sources

reinforcement learning: use of software agents to take actions in an environment in order to maximize a cumulative reward

reinforcement learning with human feedback: trains the reward model in reinforcement learning from human feedback

transfer learning: mechanism where knowledge learned from one task is reused to improve performance on a related task

transformer: deep learning transduction model that utilizes attention, weighing the influence of different parts of the input data

transpiler: translates one language into an equivalent language

NOTE: A transpiler works at a high level of abstraction. More specifically, while it may ultimately produce source code, that code is human-readable and cannot be directly executed (it requires its own compiler).

vector database: type of database that stores data as vectors

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AI	Artificial Intelligence
AIIMS	All India Institutes of Medical Sciences
ANN	Artificial Neural Network
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
BoW	Bag of Words
BSS	Business Support System
CNN	Convolutional Neural Network
CoT	Chain-of-Thought
DPR	Dense Passage Retrieval
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Fom
EBNF	Extended Backus-Naur Form
FAISS	Facebook AI Similarity Search
FB	Functional Block
FLAN	Finetuned LAnguage Net
GoT	Graph-of-Thought
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LLM	Large Language Model
LM	Language Model
LSTM	Long Short-Term Memory
MedQA	Medical Question Answering
MoE	Mixture of Experts
NEET	National Eligibility cum Entrance Test
NLP	Natural Language Processing
OSS	Operational Support System
PHP	Hypertext Preprocessor
PPO	Proximal Policy Optimization
RAG	Retrieval Augmented Generation
ReLU	Rectified Linear Unit
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network
SiC	Skills-in-Context
SOP	Sentence-Order Prediction
ToT	Tree-of-Thought
TPU	Tensor Processing Unit
TRPO	Trust Region Policy Optimization
USMLE	United States Medical Licensing Examination
VLM	Visual Language Model

4 Natural Language Processing with AI

4.1 Introduction (informative)

Natural Language Processing (NLP) is a branch of machine learning that enables machines to "understand" human language. A combination of linguistics, computer science, and machine learning, NLP works to transform regular spoken or written language into a form that can be processed by machines.

The purpose of this clause is to explain why a Transformer Architecture is preferred to other approaches.

4.2 Problems with Natural Language Processing (informative)

There are a number of fundamental problems in processing natural language. The most obvious first problem is that the amount of text to be processed is *variable*. A linear algebra model cannot deal with vectors with varying dimensions. This means that vector processing is recommended to be used. However, the size of the vectors grow as the size of the text grows (see clause 4.3.1 for a naïve solution and its attendant problems).

Word order is critical, as the ordering often provides important semantics. This causes the size of the vectors used to increase. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers (such as index terms). It is used in information filtering, information retrieval, indexing and relevancy rankings. Its first use was in the SMART Information Retrieval System.

Both of these problems dictate an increased computational cost as size increases.

4.3 Background and Previous Work (informative)

4.3.1 Bag of Words Model

The Bag of Words (BoW) model represents the input text as a vector. The size of the vector is the size of the number of terms in the text. Hence, most of the vector elements will be zeros. To minimize the size of the vector for computation, only the positions of the presented terms are stored.

Each element denotes the normalized number of occurrence of a term in the present document. BoW uses *exact* term matching to count the number of occurrences of each term.

For example, given the two sentences:

John likes to read books. Kim also likes books.
Kim also likes to watch game shows.

The corresponding BoW models are:

BoW1 = {"John":1,"likes":2,"to":1,"read":1,"books":2,"Kim":1,"also":1};

BoW2 = {"Kim":1,"also":1,"likes":1,"to":1,"watch":1,"game":1,"shows":1}.

The Bag-of-words model is an *orderless* document representation. The only thing that matters is the counts of terms. This means that simple patterns present in the text cannot be found. For example, note that in all three sentences, the verb "likes" always follows a person's name in this text.

Another critical problem with the BoW model is that it ignores the ordering of the words. For example: "Work to live" is different from "Live to Work." To keep the data order, it is necessary to increase the dimension of the graph (n-gram) to add the order into our equation.

4.3.2 n-Gram Model

The n-gram model is an alternative to the BoW model. This model is a contiguous sequence of n items from a given sample of text. The items can be phonemes, syllables, letters, words or base pairs according to the application Applying to the same example above, a bigram model will parse the text from the first sentence into the following units and store the term frequency of each unit as before.

[
"John likes",
"likes to",
"to read",
"read books",
"Kim also",
"also likes",
"likes books",
]

In n-gram models, the probability of a word depends on the (n-1) previous comments, which means that the model will not correlate with words earlier than (n-1). To overcome that, n is increased, which increases the computational complexity exponentially.

While variations of both the BoW and n-gram models exist, none provide enough improvements to make them competitive with other models below (especially transformers).

4.3.3 Recurrent Neural Networks Model

The Recurrent Neural Network (RNN) is the same as the n-gram model, except that the output of the current input will depend on the output of all of the previous computations. More specifically, an RNN has connections between nodes that form a directed or undirected graph along a temporal sequence (e.g. the ingesting of the characters of a word one character at a time), enabling it to exhibit temporal dynamic behaviour. This is shown in Figures 4.3.3-1a and 4.3.3-1b.

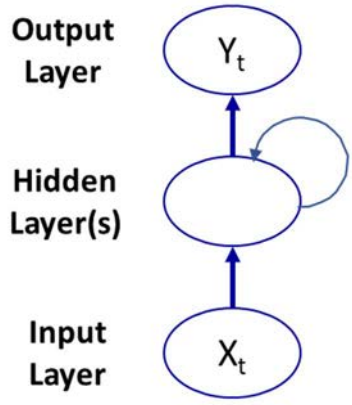

Figure 4.3.3-1a: Rolled RNN

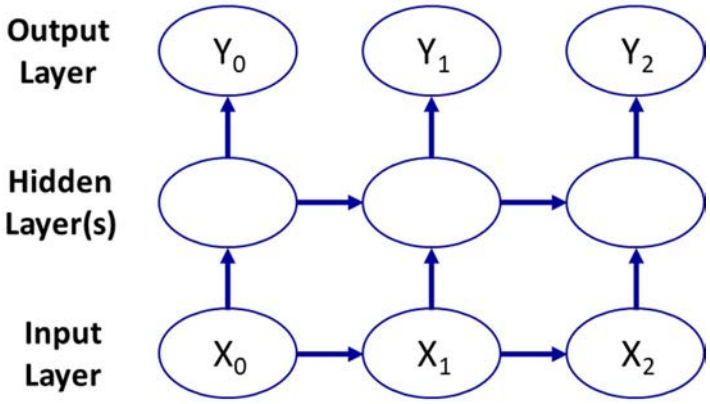

Figure 4.3.3-1b: Unrolled RNN

Figure 4.3.3-1a is a shorthand version of Figure 4.3.3-1b. The term "rolled" means that the structure is collapsed upon itself. Given a sentence, the output of the Rolled RNN is the predicted output, while the Unrolled RNN shows each individual step in determining the predicted output. The RNN handles a variable-length sequence by having a recurrent hidden state whose activation at each time is dependent on that of the previous time.

The key element of the RNN is the hidden layer, which is used to remember some information about a sequence. This enables the RNN to predict the next word of a sentence by examining the previous words that it found. This also makes the RNN fundamentally different from other artificial neural networks (which work in a linear fashion in both the feedforward and back-propagation processes), since the RNN follows a recurrent relation and uses back-propagation through time to learn (i.e. the errors at each time step are calculated to update the weights).

A neural network is a series of nodes, or neurons. The weight is the parameter within a neural network that transforms input data within the network's hidden layers. A weight can be thought of as the strength of the connection, and affects how much influence a change in the input has on the output. There is also a bias, which represents how far off the predictions are from their intended value. Within each node is a set of inputs, weight, and a bias value. As an input enters the node, it gets multiplied by a weight value and the resulting output is either observed, or passed to the next layer in the neural network. Often the weights of a neural network are contained within the hidden layers of the network. The weights and biases are learnable parameters, and are typically randomized before training begins. RNNs apply weights to the current and to the previous input. Furthermore, an RNN also adjusts the weights through gradient descent and backpropagation through time. Back-propagation fine tunes the weights of a neural net based on the error rate (i.e. loss) obtained in the previous iteration. Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

Learnable parameters in the RNN are shared in each layer in order to create a common function that can be applied at all time steps. Parameters are used to train the model. At each time step, the loss is computing and is backpropagated through the gradient descent algorithm.

The RNN consists of multiple fixed activation function units, one for each time step. Each unit has an internal hidden state, which holds the past knowledge of the network currently at a given time step. The hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation at each time step:

$$h(t) = f(x(t) + h(t-1) + b_h)$$

where h(t) is the new hidden state, h(t-1) is the old hidden state, x(t) is the current input, b_h is the bias parameter, and f() is a fixed function with trainable weights.

The activation function (i.e. a function that determines whether a neuron will be activated) uses the traditional tanh function applied to the sum of the weight times the recurrent neuron plus the weight times the input neuron. The output function is the weight of the output layer times the current state. Figure 4.3.3-2 can be simplified to emphasize their repeating structure using a single activation function, as it is done below.

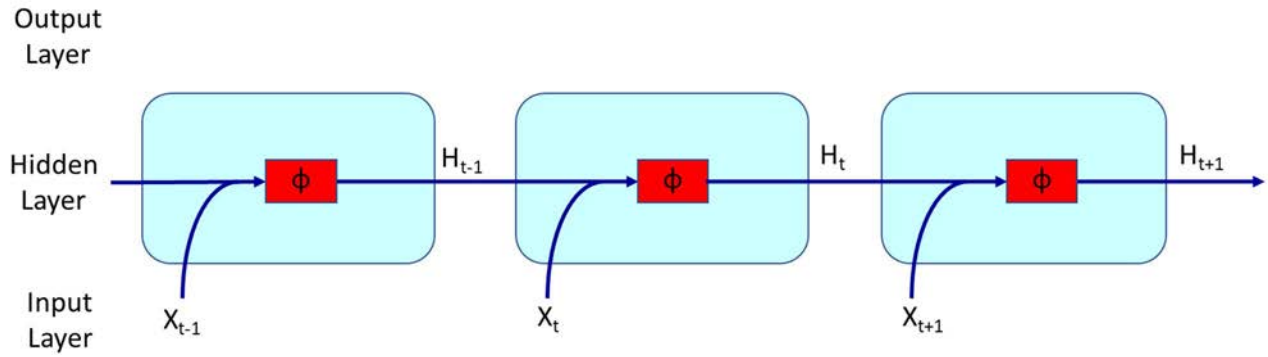

Figure 4.3.3-2: Repeating Cell in a Standard RNN

Models that are trained with gradient descent and backpropagation are subject to two problems:

1) Vanishing gradients occurs when the gradient becomes too small. When the gradient continues to become smaller, the earlier layers in the network will learn more slowly than later layers. This causes the weight parameters to continue to update until they become insignificant, which results in an algorithm that is no longer learning.

2) Exploding gradients occur when the gradient becomes too large. In this case, the model weights will grow too large, and they will eventually be represented as undefined.

Actual RNNs are not constrained to have the same number of inputs and outputs. For example, RNNs can map many inputs to one or many outputs. Bidirectional recurrent neural networks are another a variant of RNNs. Unidirectional RNNs can only use previous inputs to make predictions about the current state, but bidirectional RNNs can also use future data to improve their accuracy.

4.3.4 Convolutional Neural Networks Model

A Convolutional Neural Network (CNN) is a Deep Learning algorithm that takes an input, assign importance (learnable weights and biases) to various aspects/objects in the input, and be able to differentiate one from the other. It specializes in processing data that has a grid-like topology, such as an image. The pre-processing required in a CNN is much lower compared to other classification algorithms. A CNN can determine spatial and temporal dependencies in the input through the application of relevant filters. A convolution is the application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

CNNs have the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modelling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images. The architecture of a CNN is analogous to that of the connectivity pattern of neurons in the human brain. Individual neurons respond to stimuli only in a restricted region of the visual field; a collection of such fields overlap to cover the entire visual area. A typical CNN has three main layers: convolutional, pooling, and fully connected. The convolutional layer uses filters that perform convolution operations as it scans the input with a filter, building up a feature map. The pooling layer is typically applied after the convolution layer, and applies a particular function (e.g. minimum, average or maximum) of its current view. The fully connected layer connects each input to all of its neurons, and is used to optimize objectives.

CNNs are not applicable to word translation due to high computational cost, but some architectures do use a convolutional function as part of their architecture.

4.3.5 Long Short Term Memory Model

Long Short-Term Memory (LSTM) was created to solve the vanishing and exploding gradient problems that RNN approaches had. LSTM focuses on modelling chronological sequences and their long-range dependencies more precisely than conventional RNNs. The main difference between an RNN and an LSTM architecture is that the LSTM's hidden layer is a gated unit. The problematic issues of vanishing gradients is solved through LSTM because it keeps the gradients steep enough, which keeps the training relatively short and the accuracy high.

A simplified block diagram of an LSTM cell is shown in Figure 4.3.5-1.

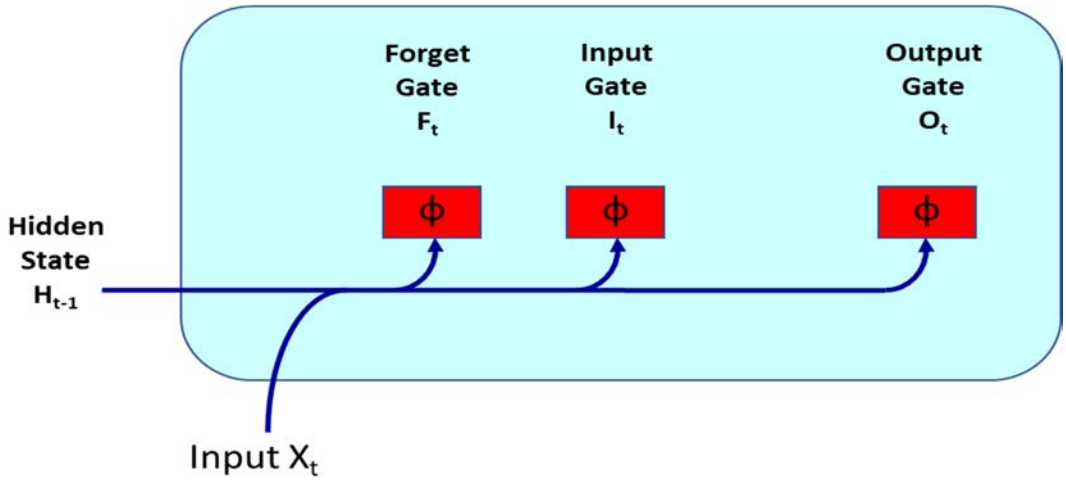

Figure 4.3.5-1: Basic LSTM Cell Architecture

The three parts of an LSTM cell shown in Figure 4.3.5-1 are known as gates. The first part is called the Forget gate, and chooses whether the information coming from the previous timestamp is to be remembered or can be forgotten. The second part is called the Input gate, and learns new information from the input to this cell. The last part is called the Output gate, and sends the updated information from the current timestamp to the next timestamp. The gate calculations are computed as follows:

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$

$$F_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

Where W_{xy} are weight parameters and b_x are bias parameters.

The next step in understanding the LSTM architecture is to add a memory cell. This is shown in Figure 4.3.5-2.

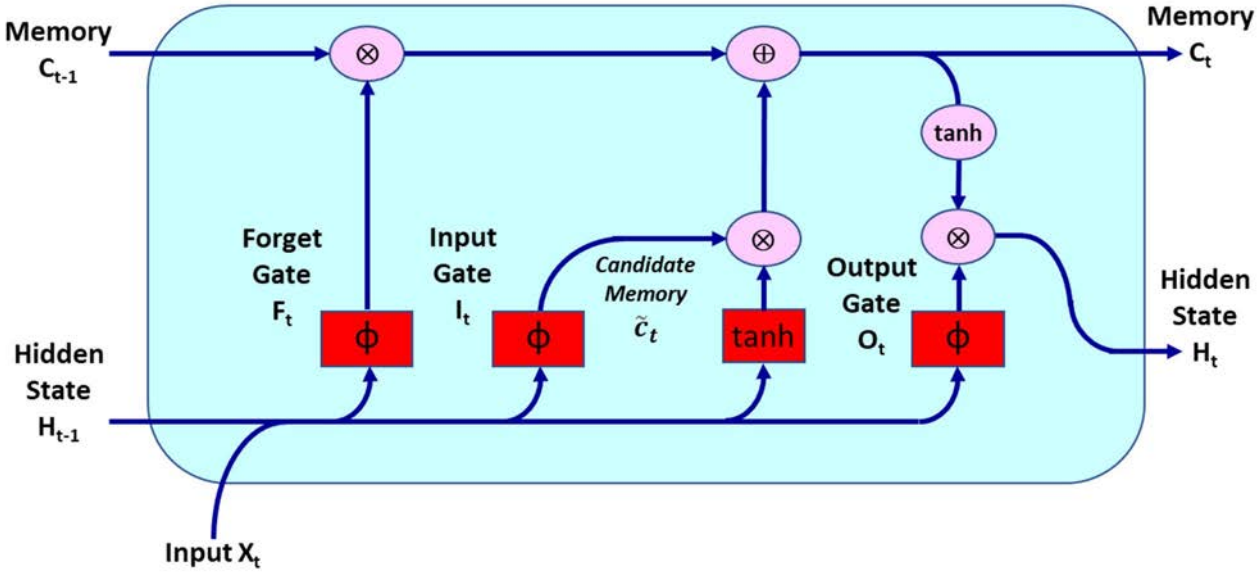

Figure 4.3.5-2: Adding Cell Memory to the LSTM

The candidate memory cell (since the input and forget gate contributions have not yet been incorporated) uses a tanh activation function:

$$\tilde{\mathbf{c}}_t = tanh(x_t \times W_{xc} + U_f + H_{t-1} \times W_{xc} + b_c)$$

where W_{xc} and W_{xc} are weight parameters and b_c is a bias parameter.

Incorporating the contributions of the input and forget gates, yields the following:

$$C_t = F_t \otimes C_{t-1} \times W_{xc} + I_t \otimes \tilde{\boldsymbol{c}}_t$$

where ⊗ denotes a point-wise multiplication.

In Figure 4.3.5-2, $C_{T, T-1}$ and $H_{T, T-1}$ are hidden and cell memory states for the current and previous time stamps. The hidden state represents short-term memory, while the cell state represents long-term memory.

The hidden state is computed as follows:

$$H_t = O_t \otimes tanh(C_t)$$

Whenever the output gate approximates 1, all memory information is passed through to the predictor, whereas when the output gate is close to 0, all information within the memory cell is retained, and no further processing is performed.

One of the advantages of this architecture is when its input contains multiple sentences. As the LSTM moves from the first sentence to the second sentence, it needs to realize that a different subject is being discussed. This is accomplished by the Forget gate. If two sentences both refer to the same subject, the input gate determines whether new information carried by the input is important or not. The function of the output gate is to predict the next term in the input sequence. The hidden state is a function of Long term memory C_t and the current output.

One weakness of the LSTM, and of many contemporary RNNs, is capacity. They are designed so that each unit of memory can influence every other unit in memory with a learnable weight. But this results in a computationally inefficient system: the number of learnable parameters in the model grows quadratically with the memory size. For example, an LSTM with a memory of size 64 KB results in parameters of size 8 GB. Circumventing this memory capacity bottleneck has been an active research area.

4.3.6 Gated Recurrent Units

Gated Recurrent Units (GRUs) are slightly simplified variants of the basic LSTM architecture, and typically provide faster computation. The key difference between RNNs and GRUs is that GRUs support dedicated mechanisms for when a hidden state needs to be updated as well as when it needs to be reset. These mechanisms are learned. For example, if the first token is very significant, the GRU will learn not to update the hidden state after the first observation. Conversely, if the first token is irrelevant, the GRU will learn to skip it.

A GRU is similar to an LSTM with a forget gate, but has fewer parameters because it does not have an output gate. Each recurrent unit of the GRU adaptively captures dependencies at different time scales. Similarly to the LSTM, the GRU has gating units that modulate the flow of information inside the unit; however, the GRU gating units do not have a separate memory cells. A GRU has two special gates, called a Reset Gate and an Update Gate. These are shown in Figure 4.3.6-1. They are both vectors whose values are in the range (0,1). The purpose of the Reset Gate ("R" in) is to control how much of the previous state needs to be remembered. The purpose of the Update Gate ("Z") is to control how much of the new state is simply a copy of the old state.

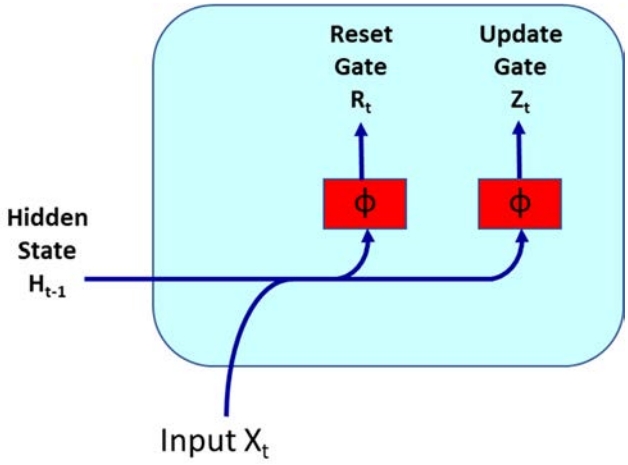

Figure 4.3.6-1: Gates in the GRU Architecture

The formulae to calculate the reset and update gate functions are:

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r)$$

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z)$$

The *candidate* hidden state is calculated as:

$$\tilde{H}_t = tanh(X_t W_{xh} + (R_t \otimes H_{t-1})\, W_{hh} + b_r)$$

This is a candidate, since the contribution of the update is included. The final hidden state is calculated as:

$$H_t = Z_t \otimes H_{t-1} + (1 - Z_t) \otimes + \tilde{H}_t$$

Whenever the update gate X_t is close to 1, the old state is kept, meaning that the information from X_t is ignored. This can be thought of as skipping time step t in the dependency chain. In contrast, whenever Z_t is close to 0, the new latent state H_t approaches the candidate latent state $\tilde{H}_t$. This leads to the final block diagram of a GRU.

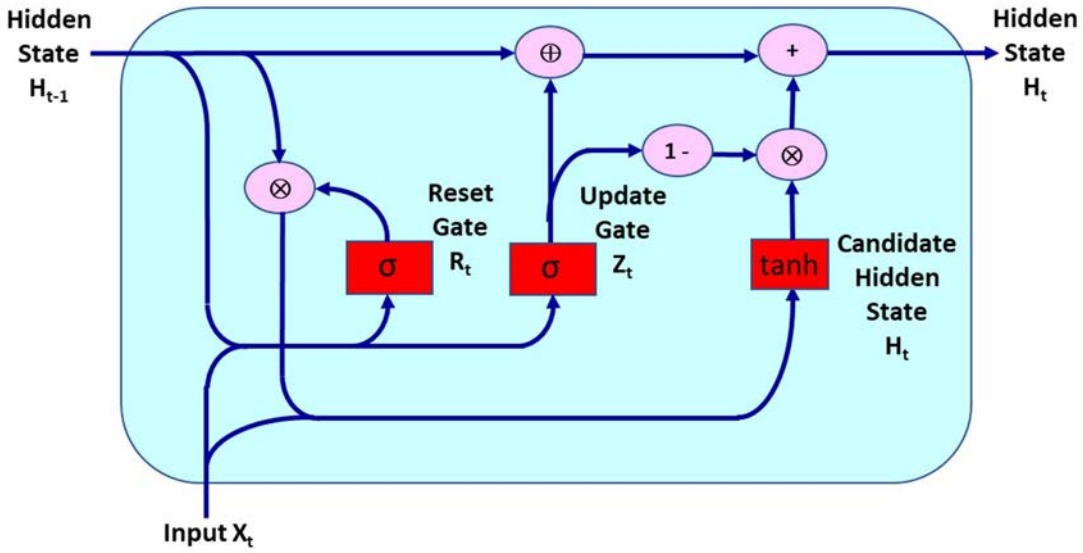

Figure 4.3.6-2: The GRU Architecture

Both the LSTM and the GRU keep the existing content of the activation and add new content to it, while the RNN replaces the activation with a new value computed from the current input and the previous hidden state. This enables both the LSTM and the GRU to remember the existence of a specific feature in the input stream for a long series of steps. Any important feature, decided by either the forget gate of the LSTM unit or the update gate of the GRU, will not be overwritten but be maintained as it is. In addition, this effectively creates shortcut paths that bypass multiple temporal steps, which allows the error to be back-propagated without vanishing too quickly.

4.3.7 Attention

4.3.7.1 Motivation

Prior to the concept of attention, NLP models would first capture all of the information in the input sentence (e.g. the details of objects, and how objects are related to each other) in an intermediate state and then use this intermediate state information to compute the output. The size of the vector used for this intermediate state is fixed.

If an input consists of very long text, the intermediate state fails because its fixed state is too small to contain all of the information. Typically, the system would "forget" the first part of the intermediate state by the time it completes processing the entire input. This yields incorrect outputs.

Hence, there are three motivations:

1) to remove the bottleneck of a fixed length vector; and

2) to relieve the intermediate state from being solely responsible for encoding all of the information available to the decoder; and

3) context for different parts of speech changes throughout a sentence. Therefore, all words need to be examined at the same time, enabling a system to learn to "pay attention" to the correct parts of the sentence depending on the task at hand.

This has given rise to what AI people call "attention". It is simply a notion of memory, gained from attending at multiple inputs through time.

4.3.7.2 Definition

The Attention mechanism [i.3] overcomes the information bottleneck of the intermediate state by allowing the decoder model to access all of the hidden states, rather than a single vector (i.e. the intermediate state) build out of the encoder's last hidden state, while predicting each output. More specifically, the information (from the intermediate state) can be spread throughout the sequence of encoder hidden states, which can be selectively retrieved by the decoder according to where in the input it is processing. This is done by using the Attention mechanism to search for a set of positions in the input text where the most relevant information is concentrated. The model then predicts next word based on context vectors associated with these source positions and all the previous generated target words.

Given a set of key-value pairs and a query, an attention mechanism computes weights of each key with respect to the query, and aggregates the values with these weights to form the value corresponding to the query. The queried values are invariant to the ordering of the key-value pairs Attention consists of an encoder that encodes a source sentence into a fixed-length vector from which a decoder generates a translation. The essence of attention is that is enables a model to automatically search for parts of a source text that are relevant to predicting a target word, without having to explicitly form these parts ahead of time.

Multi-head attention allows the model to jointly inspect information from different representation subspaces at different positions. While powerful, this requires tensor functions.

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. self-attention (where keys and queries are identical) to give expressive sequence-to-sequence mappings in natural language processing. Put another way, each query attends to all the key-value pairs and generates one attention output. Since the queries, keys, and values come from the same place, this performs self-attention. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations. End- to-end memory networks are based on a recurrent attention mechanism instead of sequence aligned recurrence and have been shown to perform well on simple language question answering and language modelling tasks.

4.3.7.3 Example

The following is a simple example of how self-attention works.

Sentence A: "I poured coffee from the carafe into the mug until it was full."
Sentence B: "I poured coffee from the carafe into the mug until it was empty."

The difference between sentence A and sentence B is that the adjective at the end of sentence a (i.e. "full") was changed to "empty" in sentence B. Self-attention recognizes that, and hence, changes the reference that "it" refers to.

There are three types of attention that a given model can contain:

- Encoder-Decoder attention: attention between the input sequence and the output sequence.
- Self attention in the input sequence: attention to all the words in the input sequence.
- Self attention in the output sequence: attention to all the words in the input sequence.

By changing one word "full" — > "empty" the reference object for "it" changed. Changes like this are difficult for an LLM to keep up with unless it is augmented with some type of grammatical analyser, such as a parser.

4.4 Transformers and Large Language Models

4.4.1 Motivation (informative)

Clause 4.3 showed some of the problems in processing natural language with previous architectures. In summary, RNNs cannot handle vanishing and exploding gradients (i.e. when the gradient becomes too small or too large). This introduced LSTM and later GRU networks to overcome these problems by using memory cells and gates. However, long-term dependencies are not addressed because both architectures rely on these new gate/memory mechanisms to pass information from old steps to the current ones. The solution was to use self-attention (see clause 4.3.7) to process an entire sentence, as opposed to process individual words sequentially. Self-attention enables the system to focus on parts of the input sequence while the output sequence is being predicted. This enables the transformer to have access to all of the words in the sentence. More importantly, the self-attention mechanism makes sure each word is related to all of the words in the sentence. This is described in much more detail in [i.2].

Another important motivation is parallelization. In previous architectures, each hidden state has dependencies on the previous words' hidden state. Thus the word embeddings (i.e. a vector of words, where words with closer meanings are closer to each other) of the current step are generated one time step at a time. In contrast, a Transformer architecture has no concept of time; the input sequence can be passed into the Encoder as a single term.

A Large Language Model (LLM) is a statistical model that predicts the next word in a sequence. It is trained on a massive dataset of text, and can be used to generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way. The majority of LLMs are architected as variants of the Transformer architecture. The present document will only consider these types of LLMs.

LLMs are trained on massive datasets of text, typically in the order of billions of words. This allows them to learn the statistical relationships between words and phrases, and to generate text that is both grammatically correct and semantically meaningful.

All LLMs are generative: they can create new content, such as text, images, or music, by learning the patterns and structures of existing data and then using those patterns to generate new data that is similar to the original data. However, not all transformers are generative, because they are only used for tasks that do not require generation, such as classification or question answering. For example, the BERT model (see clause 4.4.4.4) is a transformer model that is used for natural language understanding tasks, and is not used for text generation or translation.

4.4.2 Basic Transformer Architecture (informative)

4.4.2.1 Architectural Overview

The transformer revolutionised machine learning, as it replaced recurrent and convolution mechanisms and architectures with an attention mechanism. They were first introduced in [i.2].

The Transformer architecture takes the encoder-decoder models (see clause 4.3.7). The key difference is that instead of attention being one of the mechanisms used by these models, in the Transformer architecture, attention was the only mechanism used to derive dependencies between input and output.

There are an equal number of encoders and decoders; the number is chosen based on the type of task. This is shown in Figure 4.4.2-1.

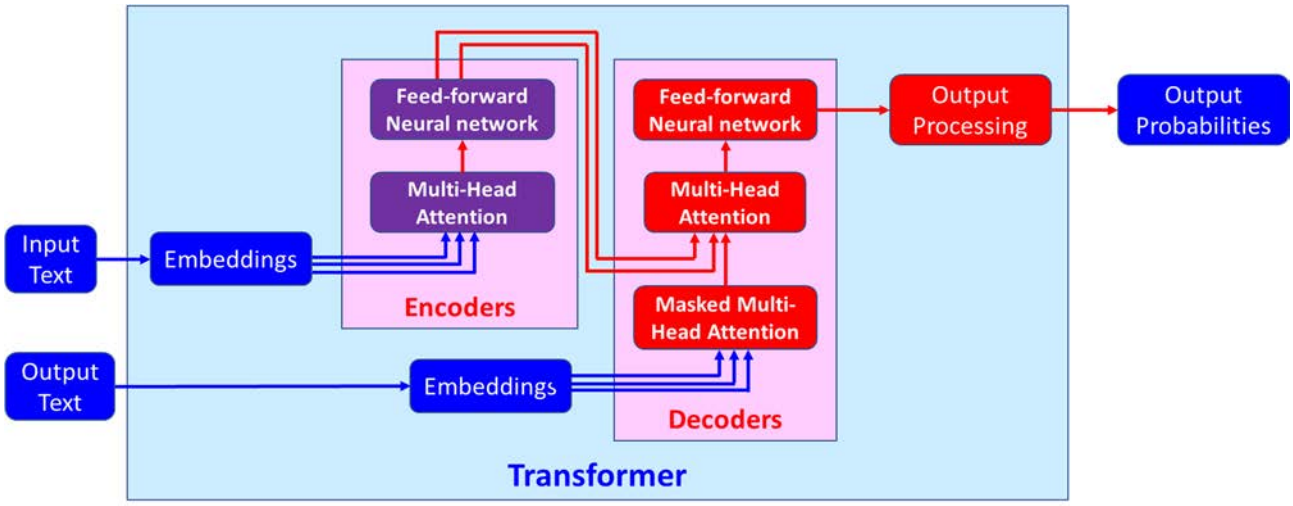

Figure 4.4.2-1: Basic Transformer Architecture

Input text is fed to the Encoder, which converts each word in the input to an embedding (i.e. a high-dimensional vector of words) to represent the meaning of each word. Word embedding on its own lacks any positional information (remember, this was provided by RNNs since they process input text sequentially. To preserve the positional information, the transformer injects a vector into individual input embeddings. This vector follows a specific periodic function (e.g. a combination of various sine and cosine functions having different frequencies) that the model learns and is able to determine the position of individual word with respect to each other based on the values. This injected vector is called "positional encoding" and is added to the input embeddings at the entrance to both encoder and decoder stacks.

Each encoder consists of two layers: a self-attention layer and a feed-forward neural network layer. The encoder's inputs first flow through a self-attention layer - a layer that helps the encoder look at other words in the input text as it encodes a specific word.

The encoder maps an input sequence of text $(x_1, \ldots, x_n)$ to an intermediate sequence of text $z = (z_1, \ldots, z_n)$. This can be conceptualized as the output from self-attention with some additional post-processing. More specifically, this layer outputs word vectors with positional information; that is the word's meaning and its context in the sentence. Since an attention vector for each word could be weighted too highly, a *multi-head* attention approach is used, where multiple attention vectors are used per word and a weighted average (or some other means to normalize the result) is taken to compute the final Attention vector for every word. Multiple attention heads allow the model to jointly attend to information from different representation sub-spaces at different positions; this provides a better contextual meaning, and is not possible using a single attention head.

The outputs of the self-attention layer are fed to a feed-forward neural network in parallel. The exact same feed-forward network is independently applied to each position. The output will be a set of encoded vectors for every word.

Each decoder consists of three layers: a multi-head *masked* self-attention layer, a multi-head attention layer, and a feed-forward neural network layer. The encoder-decoder multi-head attention layer and the feed-forward neural network layer of the decoder are the exact same as those for the encoder; the difference is that the decoder also has a *masked* multi-head attention layer preceding it. The number of encoder and decoder units is a hyperparameter (the original paper [i.2] used 6). As shown in Figure 4.4.2-1, the masked multi-head self attention unit is based on the word embeddings of the output. The masked multi-head attention block operates similarly to the multi-head attention in the encoder block, except that only the previous words of the solution text are fed into this decoder's attention block. The masked attention block computes the Attention vectors for current and prior words. This, along with some of the outputs of the encoder, feed a multi-head attention mechanism. The multi-head attention block in the decoder acts as the encoder-decoder, which receives vectors from the encoder's multi-head attention and decoder's masked multi-head attention blocks. This attention block will determine how related each word vector is with respect to each other, and this is where the mapping from input word to output word happens. The output of this block is a set of attention vectors for every word, where each vector represents the relationships with other words in input and output. These vectors are passed into the feed-forward layer linear layer. Like the encoder's feed-forward layer, this layer normalizes each word consisting of multiple vectors into a single attention vector for the next decoder block or a linear layer. The purpose of the decoder is to predict the following word, so the output size of this feed-forward layer is the number of words in the vocabulary. Softmax transforms the output into a probability distribution, which outputs a word corresponding to the highest probability of the next word.

At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

4.4.3 Basic Transformer Models (informative)

4.4.3.1 Introduction

The Transformer uses multi-head attention in three different ways:

1) In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as described in [i.37].

2) The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.

3) Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. Leftward information flow in the decoder needs to be prevented to preserve the auto-regressive property. This is implemented inside of scaled dot- product attention by masking out (setting to) all values in the input of the softmax which correspond to illegal connections.

4.4.3.2 The Original Transformer Architecture

Figure 4.4.3.2-1 shows the original transformer architecture proposed in [i.2].

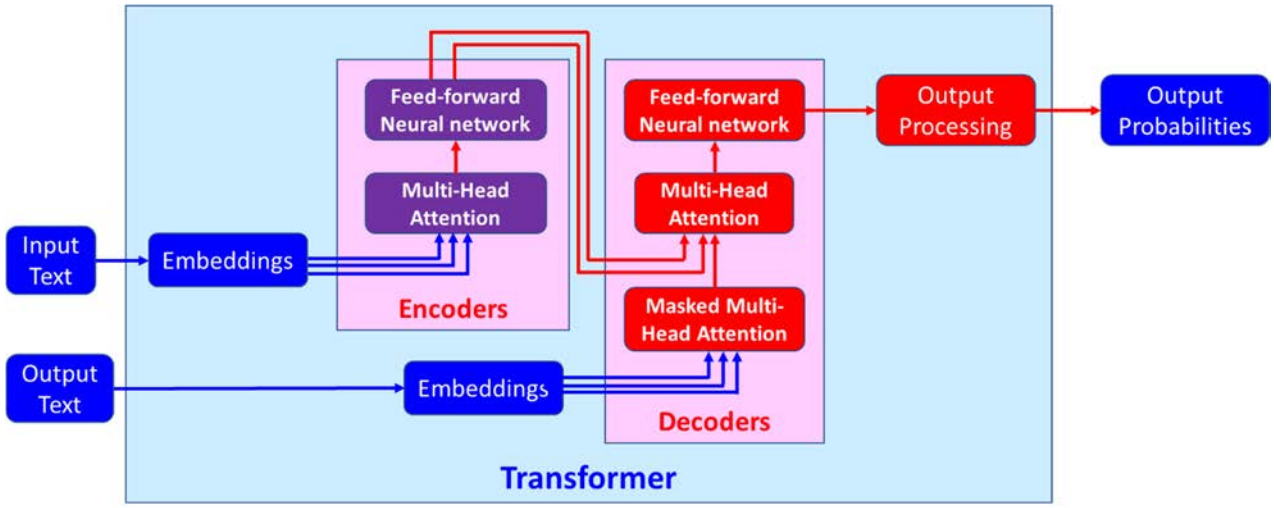

Figure 4.4.3.2-1: Original Transformer Architecture

The following new concepts are introduced by [i.2]:

Residual Connections. The encoder is composed of a stack of 6 identical layers. Each layer has two sub-layers: a multi-head self-attention mechanism, and a position-wise fully connected feed-forward network. A residual connection uses two or more skip connections to jump over one or more layers. The skip connection skips the activation function of the layers that it skips, and instead uses a rectifier or ReLU (Rectified Linear Unit) activation function, which is:

$$function: f(x) = max(0,x),$$

where x is the input to a neuron and f(x) is the output. This is also known as a ramp function.

NOTE: This characteristic allows ReLU to introduce non-linearity into neural networks, which is crucial for their ability to learn complex patterns in data.

As can be seen, each of the units in the encoder has a residual connection, followed by layer normalization. That is, the output of each sub-layer is LayerNorm (x + Sublayer(x)), where Sublayer (x) is the function implemented by the sub-layer itself. The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers that are identical to the encoder sub-layers, the decoder inserts a third sub-layer, which performs multi-head attention over the output word embeddings offset by one position. The self-attention sub-layer is modified by masking to ensure that the predictions for position i can depend only on the known outputs at positions less than i.

Multi-head Attention. Multi-head attention is based on three matrices, called Q (Query), K (Key), and V (Value). These vectors are trained and updated during the training process. The idea is to map a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. This is done by taking the dot product of the Query vector for a given word with the keys vectors of all the words. Then, these scores are divided by the square root of the dimension of the key vector and are normalized using the softmax activation function. These normalized scores are then multiplied by the value vectors, and the resultant vectors are summed to provide the final vector. This is the output of the self-attention layer. It is then passed on to the feed-forward network as input. Self-attention is thus computed not once but multiple times in the Transformer's architecture, in parallel and independently. The outputs are concatenated and linearly transformed.

4.4.3.3 Transformer Advantages

There are several advantages of attention layers over recurrent and convolutional networks, the two most important being:

1) their lower computational complexity;

2) their higher connectivity, especially useful for learning long-term dependencies in sequences; and

3) their ability to quickly adapt to other tasks that they have not been trained on.

Transformers have a lower computational complexity than RNNs or CNNs because they were designed to avoid recursion and allow parallel computation, which reduces training time and improves performance on long dependencies. The main characteristics of transformers are that they are non-sequential, meaning that sentences are processed as a whole rather than word by word, and they use self-attention to compute similarity scores between words in a sentence. They instead process a sentence as a whole. That is why there is no risk to lose (or "forget") past information. They also use positional embeddings to encode information related to the specific position of a token in a sentence. In addition, they use multi-head attention and positional embeddings both provide information about the relationship between different words.

RNNs and their variants only have feedforward connections, impairs the learning of long dependencies. In contrast, transformers use self-attention mechanisms to compute similarity scores between all pairs of elements in a sequence, allowing them to capture long-range dependencies more effectively. This is because self-attention allows the model to directly access the representations of all elements in the sequence, regardless of their distance in time.

Pretrained transformer models can adapt extremely easily and quickly to tasks they have not been trained on because of their ability to transfer knowledge from one task to another (this is known as transfer learning). This is possible because transformers are designed to learn context and meaning by tracking relationships in sequential data. By finding patterns between elements mathematically, transformers eliminate the need for explicit feature engineering. In addition, the math that transformers use lends itself to parallel processing, so these models can run fast.

4.4.3.4 Transformer Limitations

There are four major limitations to this basic transformer architecture:

- First, attention can only deal with fixed-length text strings. The text has to be split into a certain number of segments or chunks, before being fed into the system as input.

- Second, this chunking of text causes context fragmentation. For example, if a sentence is split from the middle, then a significant amount of context is lost. In other words, the text is split without respecting the sentence or any other semantic boundary.

- Third, the use of self-attention with every other token in the input means that the processing will be in the order of O(N^2, which means that it is going to be costly to apply transformers on long sequences, compared to RNNs.

- Fourth, basic Transformer architectures cannot process input sequentially. This means that Transformers cannot perform recursive computation on sequential tasks. The number of transformations possible on the input is bounded by the model depth.

4.4.4 Other Transformer Models (informative)

4.4.4.1 Introduction

The following major types of Transformer models have been created to address the limitations discussed in clause 4.4.3.3 along with other limitations that will be described below.

4.4.4.2 Transformer-XL

Transformer-XL [i.4] was specifically created by Google® to overcome the two issues described in clause 4.4.3.3, and consists of two techniques: a segment-level recurrence mechanism and a relative positional encoding scheme. This enables learning dependency beyond a fixed length without disrupting temporal coherence. Transformer-XL is open source, and uses the Apache® 2.0 license.

More specifically, recurrence is introduced a self-attention network. Instead of computing the hidden states from scratch for each new segment, Transformer-XL reuses the hidden states from previous segments. During training, the hidden state sequence computed for the previous segment is fixed and cached to be reused as an extended context. In each segment, each hidden layer receives the output of the previous hidden layer and the output of the previous segment. The use of contextual information from multiple previous segments increases the ability to capture long-range dependency information. Hence, this solves the context fragmentation issue and without the evaluation speed by processing an entire segment while using the representations from the previous segments without recomputation.

This approach also solves a subtle issue. If recurrence is used without any corrective information, then the resulting positional information will become incoherent, since tokens from different segments will end up with the same positional encoding (this is called *temporal confusion*). Transformer-XL solves this problem by using relative positional encodings by encoding positional information bias *in the hidden states themselves*. This is different from other approaches, which typically incorporate bias in the initial embedding. The use of fixed embeddings with learnable transformations makes it more intuitive and more generalizable to longer sequences. The relative positional encodings make segment-level recurrence possible so that Transformer-XL can model much longer-term dependency than a vanilla Transformer model.

One last important feature of Transformer-XL is that the size of the memory can be used to control the size of past activations, thus controlling how much context is kept. One last important feature of Transformer-XL is that the size of the memory can be used to control the size of past activations, thus controlling how much context is kept.

4.4.4.3 Compressive Transformers

Another drawback of the original transformer architecture is the computational cost of attending to every time-step, as well as the associated storage cost of preserving this memory. Recall that the transformer represents the past as a tensor of (depth x memory size x dimension) of past observations. Furthermore, a human retains memories over a diverse array of timescales, from minutes to months to years to decades, based on *context*.

The Compressive Transformer [i.5] is a simple extension to the Transformer that maps past hidden activations (memories) to a smaller set of compressed representations (compressed memories). The Compressive Transformer uses the same attention mechanism over its set of memories and compressed memories, learning to query both its short-term granular memory and longer-term coarse memory.

The Compressive Transformer builds on the ideas of the Transformer-XL, which maintains a memory of past activations at each layer to preserve a longer history of context. Instead of discarding past activations when they become sufficiently old, the Compressive Transformer compacts these old memories, and store them in an additional compressed memory. Different functions can be chosen as the compression mechanism, where each function has a different compression factor and associated loss. The loss guides the Transformer to keep around task-relevant information. It can learn to filter out irrelevant memories, as well as combine memories so that the salient information is preserved and retrievable over a longer period of time.

The Compressive Transformer is able to produce narrative in a variety of styles, from multi-character dialogue, first-person diary entries, or third-person prose. Although the model does not have an understanding of language that's grounded in the real world, or the events that take place in it - by capturing longer-range correlations, the emergence of more coherent text is made possible.

One advantage of the Compressive Transformer is its increased ability to use past activations to find long-range temporal dependencies efficiently. This will be especially important when long-range language modelling (e.g. books or certain types of conversational agents) is important. Another advantage is that compression appears to be more suitable for scaling memory and attention that, for example, dynamic or sparse attention, as these typically require custom kernels to make them efficient. In contrast, the Compressive Transformer can use simple neural network components, such as convolutions, to build effective compression modules.

The disadvantage is the additional complexity of the architecture for those tasks that do not require long-range reasoning.

4.4.4.4 BERT Models (informative)

4.4.4.4.1 The Original BERT Model

A language model is a probability distribution over words or word sequences. This does not refer to grammatical validity. Rather, it means that it defines the probability of a word or set of words occurring in a given document. As such, the purpose of BERT is different than that of a generic Transformer, whose goal is to produce output text. BERT is open source, and uses the Apache® 2.0 license.

BERT (Bidirectional Encoder Representations from Transformers) [i.6] is a new type of open source language model developed and released by Google® in late 2018, and is focussed on natural language processing tasks, including Natural Language Inference, Text Classification, Question Answering, and Named Entity Recognition. BERT achieved state-of-the-art results on a range of NLP tasks while relying on unannotated text drawn from the web, as opposed to a language corpus that's been labelled specifically for a given task. BERT is significant because it used transfer learning to pre-train a transformer encoder over unlabelled data using masking (i.e. replacing certain words in a sentence with a "[MASK]" token and then trying to predict them; this can also be applied to groups of words or even sentences).

Recall that a Transformer includes two separate mechanisms - an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. BERT's key innovation is using the encoder portion of a Transformer for bidirectional training. In contrast, previous efforts were limited to analysing a text sequence either from left to right or combined left-to-right and right-to-left.

BERT is designed to pretrain deep bidirectional representations from unlabelled text by taking both the left and the right context into account in all layers. As a result, the pre-trained BERT model can be finetuned with just one additional output layer. The input to BERT can be a single sentence or a sentence pair (as in a Question Answer task).

There are two phases to using BERT: pre-training and fine-tuning. During pre-training, the model is trained on unlabelled data over different pre-training tasks. This phase consists of two unsupervised predictive tasks, Masked Language Model and Next Sentence Prediction. Masked Language Model is a process in which some percentage (15 % in the original paper) of the input tokens are masked at random, and then those masked tokens are predicted. The final hidden vectors corresponding to the mask tokens are fed into an output softmax over the vocabulary. This represents the context of the left and right sides of the fusion, which makes it possible to pre-train the deep two-way Transformer. A downside is that there is a mismatch between pre-training and fine-tuning, since the [MASK] token does not appear during fine-tuning. To mitigate this, the authors propose a training data generator that chooses 15 % of the token positions at random for prediction. If the i-th token is chosen, it is replaced with:

1) the [MASK] token 80 % of the time;

2) a random token 10 % of the time;

3) the unchanged i-th token 10 % of the time.

The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model converges slower than directional models, a characteristic which is offset by its increased context awareness.

Tokenizers have to deal with punctuation, spaces, and other grammatical forms. Different Transformers use different tokenizers. The BERT tokenizer is called WordPiece, which deals with grammatical tokens by finding the symbol pair whose probability divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs. For example, if the two symbols are "l" and "y", they will be merged into "ly" only if the probability of "ly" divided by "l" + "y" are greater than for any other symbol pair.

The Next Sentence Prediction task enables the relationship between two sentences to be understood. This is required for some tasks, such as Question Answering, and is not directly represented in normal language modelling. Given two sentences S_1 and S_2, and given that 50 % of the time S_2 is the next sentence after S_1 (the other 50 % of the time S_2 is randomly selected from the corpus), learn the correlation.

NOTE: The steps for doing this are beyond the scope of the present document.

For fine-tuning, the BERT model is first initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labelled data from the downstream tasks. Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters. There is minimal difference between the pre-trained architecture and the final downstream architecture due to BERT's unified architecture. During fine-tuning, all parameters are fine-tuned.

4.4.4.4.2 RoBERTa

RoBERTa (Robustly optimized BERT approach) [i.7] is another open source variation that posits that the original BERT model was significantly undertrained. Hence, RoBERTa modifies key BERT hyperparameters, along with increasing the size of the data:

1) Train the model longer over more data using larger batches (from 16 GB to over 160 GB).
2) Removed the Next Sentence Prediction pre-training task.
3) Trained on longer sequences (each input is up to 512 tokens and consists of full sentences).
4) Dynamically changing the masking pattern in pretraining.

RoBERTa emphasized how data size, training time, and pretraining objectives influence the model performance. The original BERT implementation performed masking once during data pre-processing, resulting in a single static mask. To avoid using the same mask for each training instance in every epoch, training data was duplicated 10 times so that each sequence is masked in 10 different ways over the 40 epochs of training. Thus, each training sequence was seen with the same mask four times during training. RoBERTa uses this dynamic masking, where the masking pattern is generated every time new input is sent to it.

RoBERTa also changes the tokenizer from WordPiece to Byte-Pair Encoding. This relies on a pre-tokenizer that splits the training data into words according to a set of rules. After pre-tokenization, a set of unique words has been created, along with an associated frequency of each word. Next, it creates a base vocabulary consisting of all symbols that occur in the set of unique words. Finally, it learns merge rules to form a new symbol from two symbols of the base vocabulary. It does so until the vocabulary has attained the desired vocabulary size, which is a hyperparameter. RoBERTa is open source, and uses the Apache® 2.0 license.

4.4.4.4.3 ALBERT

The motivation for ALBERT (A Lite BERT) [i.8] is to lower memory consumption and increase the training speed of BERT. This is because previous attempts were focussed on increasing model size; however, at some point this becomes difficult due to GPU/TPU memory limitations and longer training times. This is also open source.

Albert decided that the WordPiece embedding used in BERT and RoBERTa was suboptimal. First, WordPiece embeddings are intended to learn context-independent representations, whereas hidden-layer embeddings are meant to learn context-dependent representations. BERT associates the size of the embedding E with the size of the hidden layer H ($E \equiv H$). Since BERT uses context, this does not make sense. Second, NLP typically requires a large vocabulary size V. However, if $E \equiv H$, increasing H means that the embedding matrix (which is V x E) becomes huge. Worse, most of the parameters will only be sparsely updated during training.

In ALBERT the embedding parameters are decomposed into two smaller matrices. Instead of projecting the one-hot vectors directly into the hidden space, they are first projected into a lower dimensional embedding space of size E, and then projected into the hidden space. By using this decomposition, the embedding parameters are reduced from $O(V \times H)$ to $O(V \times E + E \times H)$. This parameter reduction is significant when H is much larger than E.

ALBERT also eliminates Next Sentence Prediction. It posits that this task, as formulated in BERT, conflates topic prediction and coherence prediction in a single task. In its place, it proposes a loss function based on coherence. ALBERT uses a Sentence-Order Prediction (SOP) loss, which avoids topic prediction and instead focuses on modelling inter-sentence coherence. The SOP loss uses as positive examples the same technique as BERT (two consecutive segments from the same document), and as negative examples the same two consecutive segments but with their order swapped. This forces the model to learn finer-grained distinctions about discourse-level coherence properties.

ALBERT has less hyperparameters than the large version of BERT, and achieves significantly better results. However, it is computationally more expensive due to its larger structure. ALBERT is open source, and uses the Apache® 2.0 license.

4.4.4.4.4 CodeBERT

As the name implies, CodeBERT [i.31] is optimized for coding descriptions, queries, and generation. It uses the same architecture as RoBERTa, with 125 million parameters. It defines a bimodal pretrained model for programming language and natural language. This enables it to support applications such as natural language code search, code documentation, and code generation. Natural language is a sequence of words, and code is a sequence of tokens.

This model will require further review. While it is very good at coding, users also need to be able to enter natural language and have that translated to a policy. CodeBERT could be used with a DSL, but it would be difficult to use the models in [2] with it.

4.4.4.4.5 Additional Optimization Features

The above three optimizations are some of many proposed for BERT. Other possibilities include:

1) sparse factorizations of the attention matrix;
2) recomputation of attention matrices to save memory;
3) fast attention kernels for training;
4) block attention (splitting the input into blocks, apply intra self-attention to each block, then apply inter-self-attention to all blocks);
5) hard example mining (n-gram model that learns precise syntactic and semantic word relationships);
6) more efficient language modelling training;
7) exploiting redundancy inherent in the word vectors within the BERT model.

4.4.4.5 GPT Models (informative)

4.4.4.5.1 Introduction

GPT, or Generative Pre-trained Transformer, is a type of transformer-based neural network that is used for generating human-like text from given inputs. It uses a language model that is pre-trained on large datasets of text to generate realistic outputs based on user prompts. GPT is built using only transformer decoder blocks (no encoder blocks). GPT models do not use encoder blocks because they are designed to generate text one token at a time, using only the context of the previous tokens in the sequence. This is known as an auto-regressive model. The encoder part of the original transformer architecture is not needed in GPT models because the model does not need to learn the representation of the input sequence. Instead, GPT models use a masked self-attention mechanism that only looks at prior tokens.

NOTE: This means that such models cannot plan or reason (whereas transformers can do both) without external aids (e.g. prompting).

4.4.4.5.2 OpenAI GPT Models

GPT models were popularized by OpenAI. There are four main releases: GPT-1, GPT-2, GPT-3, and GPT-4. None of these models are open source.

In GPT-1 [i.9], the multi-head attention module with cross-attention is removed from the decoder of transformer because there is no encoder in GPT. Hence, the decoder blocks used in GPT have only positional encoding, masked multihead self-attention module and feedforward network with their adding, layer normalization, and activation functions. GPT's attention mechanism focuses on word, pairs of words, pairs of pairs of words, and so on, only on the previous (left) words, pairs of words, etc. This is because the objective of GPT is to predict the next word given all of the previous words.

GPT-2 [i.10] and GPT-3 [i.11] are extensions of GPT-1 with an increased number of stacks of decoders. Hence, they have more learnable parameters and can be trained with more data for better language modelling and inference. For example, Table 4.4.4.5.2-1 lists 5 important metrics for the four GPT models. The phrase "Not released" means that OpenAI did not release these data.

Table 4.4.4.5.2-1: Metrics for OpenAI's GPT Models

	GPT-1	GPT-2	GPT-3	GPT-4
Parameters	117 million	1.5 billion	175 billion	1,76 trillion (estimated)
Decoder Layers	12	48	96	Not released
Hidden Layers	768	1 600	12 288	Not released
Context (Token Size)	512	1 024	2 048	8 192 or 32 768
Batch Size	64	512	128	1 024 and 4 096

GPT-3 introduced a number of important improvements:

- First, it used in-context learning. This mechanism improves In-context learning is a technique used to LLM performance by incorporating additional context during the training phase. This takes advantage of an LLM's ability to recognize patterns in data, which helps them minimize loss. During in-context learning, the LLM is given a prompt consisting of a list of input-output pairs that demonstrate a task. At the end of the prompt, a test input is appended and the LLM makes a prediction just by conditioning on the prompt and predicting the next tokens. Significantly, no parameters need to be optimized.

- Second, it concretises the notion of larger models performing better in zero-, one, and few-shot learning (see clause 4.5.2.1). Of course, larger context windows enable more examples to be provided.

- Third, GPT-3 was trained on a mix of five different corpora (Common Crawl, WebText2, Books1, Books2 and Wikipedia).

InstructGPT [i.12] utilizes instruction tuning to control the language model and generate desired human-like content. Instruction tuning works by first pre-training an LLM on a massive dataset of text and code. This allows the model to learn the statistical relationships between words and concepts. The pre-trained model is then fine-tuned on a collection of natural language instructions. These instructions describe how to perform different tasks, such as question answering, sentiment analysis, and text summarization. Significantly, instruction tuning helps the LLM to learn how to interpret and follow natural language instructions. This allows the LLM to perform these tasks even if it has not been explicitly trained on them. Hence, InstructGPT starts with collecting a dataset of labelled demonstrations of the desired model behaviour, which are then used as instructions to fine-tune GPT-3. This allows for controlling the model to generate answers that align with human expectations. In term of the optimization algorithm, InstructGPT leverages reinforcement learning from human feedback [i.13]. This dramatically improved the output of GPT-3.

GPT-4 [i.14] has undergone significant improvements, allowing it to perform a wide range of tasks without the need for additional training. Unfortunately, very little technical information is available. However, [i.14] makes it apparent that GPT-4 is a multimodal model, meaning that it can process both images and text. This allows it to describe humour in unusual images, summarize text from screenshots, and even answer exam questions that contain diagrams.

There is also a code-specific form, called CodeGPT. However, it was only trained on code, and hence, cannot be used for the purposes of ENI.

4.4.4.5.3 BLOOM

BLOOM [i.15] was coordinated by BigScience, an open research consortium. Over 1 200 participants from 38 countries participated. The BLOOM architecture is based on causal-decoder transformer models. BLOOM introduced a couple of key innovations to standard causal-decoder models. It is currently the largest open source LLM. BLOOM uses the RAIL license, and requires that its users comply with certain ethical and safety guidelines. Its main features include:

1) ALiBi Positional Embeddings. The researchers wanted to extrapolate to longer sequences, so instead of adding positional information to the embedding layer, ALiBi directly weakens the attention scores based on the distance between the keys and queries. A consequence of this was that it also led to smoother training and improved downstream performance, even outperforming both learned and rotary embeddings.

2) Embedding LayerNorm. The team experimented with an additional layer normalization placed immediately after the embedding layer. BigScience decided to train BLOOM with an additional layer normalization after the first embedding layer to avoid training instabilities using bfloat16 (instead of float16). Since then, float16 has been identified as the cause of many observed instabilities in training LLMs, and it is possible that bfloat16 alleviates the need for the embedding LayerNorm.

3) Megatron-DeepSpeed20 [i.38]. This is a combination of the DeepSpeed learning optimization library and Megatron-LM, a large and powerful transformer model. This framework enables three types of parallelism:

    a) Data parallelism, which replicates the model multiple times and place search replica on a different device, where it is fed a slice of the data. The processing is done in parallel, and all model replicas are synchronized at the end of each training step.

    b) Tensor parallelism, which partitions individual layers of the model across multiple devices. This places shards of this tensor on separate GPUs, making it possible to perform horizontal parallelism or intra-layer model parallelism.

    c) Pipeline parallelism, which splits the model's layers across multiple GPUs so that only a fraction of the layers of the model are placed on each GPU.

4) Large corpus. BLOOM was trained on the ROOTS corpus, which includes 498 HuggingFace datasets that cover 46 languages and 13 programming languages. The training process includes data sourcing and processing stages.

BLOOM is the first multilingual LLM trained in complete transparency. It is an open-access model, meaning that researchers, academics, non-profits, and smaller companies can download, run, and study it to investigate the performance and behaviour of recently developed large language models.

4.4.4.5.4 PaLM Models

Version 1 of the Pathways Language Model (PaLM) [i.16] was released in April 2022 and finalized in October 2022. It uses a proprietary license, and does not grant users the right to redistribute or modify the PaLM models.

PaLM version 1 is a 540-billion parameter, dense decoder-only model trained with the Pathways system. Pathways [i.17] is a large scale orchestration layer for accelerators that is designed to enable exploration of new systems and ML research ideas, while retaining state of the art performance for current models. Pathways uses a sharded dataflow graph of asynchronous operators that consume and produce futures, and efficiently gang-schedules heterogeneous parallel computations on thousands of accelerators while coordinating data transfers over their dedicated interconnects.

PaLM used the Pathways system to train a single model across 6 144 Tensor Processing Unit (TPU) pods. TPUs are custom silicon for machine learning training. The training is scaled using data parallelism at the Pod level across two TPU v4 pods, while using standard data and model parallelism within each Pod. It features the attention and feedforward layers to be computed in parallel. PaLM was trained using a combination of English and multilingual datasets that include high-quality web documents, books, Wikipedia, conversations, and GitHub code. PaLM uses a "lossless" vocabulary that preserves all whitespace (especially important for code), splits out-of-vocabulary Unicode characters into bytes, and splits numbers into individual tokens, one for each digit.

PaLM-E [i.18] a multi-modal embodied Visual Language Model (VLM) with 562 billion parameters that integrates vision and language for robotic control. It can perform a variety of tasks without the need for retraining. PaLM-E can generate a plan of action for a mobile robot platform with an arm and execute the actions itself by analysing data from the robot's camera without needing a pre-processed scene representation. This eliminates the need for a human to pre-process or annotate the data and allows for more autonomous robotic control. PaLM-E is a next-token predictor based on PaLM. PaLM-E takes continuous observations, like images or sensor data, and encodes them into a sequence of vectors that are the same size as language tokens. This enables PaLM-E to process sensory information in the same way it processes language.

PaLM 2 [i.19] provides significant advantages over PaLM. First, it has been trained on over 100 spoken-word languages. This enables it to understand nuanced phrases in different languages including the use of ambiguous or figurative meanings of words rather than their literal meaning. Second, PaLM 2 provides stronger logic, common sense reasoning, and mathematics than previous models. PaLM 2 also understands, generates and debugs code and was pretrained on more than 20 programming languages. Alongside popular programming languages like Python™ and JavaScript, PaLM 2 can also handle older languages like Fortran. These three features are based on improved dataset pretraining that extends across hundreds of languages and domains, demonstrating that larger models can handle disparate non-English datasets without sacrificing English-language understanding performance. PaLM 2 also includes control tokens to enable control over toxicity at inference time; and multilingual out-of-distribution "canary" token sequences that are injected into the pretraining data to provide insights on memorization across languages.

Med-PaLM 2 [i.20] is a medical language model that is based on PaLM 2 and is fine-tuned on medical information. Med-PaLM 2 is designed to provide high-quality answers to medical questions, and has been aligned to the medical domain and evaluated using medical exams, medical research, and consumer queries. [i.34] introduces the MultiMedQA benchmark, which was instrumental in training various PaLM models including Med-PaLM2. Med-PaLM 2 was the first LLM to perform at an "expert" test-taker level performance on the MedQA dataset of US Medical Licensing Examination (USMLE)-style questions, reaching 85 % + accuracy, and it was the first AI system to reach a passing score on the MedMCQA dataset comprising Indian AIIMS and NEET medical examination questions, scoring 72,3 %.

4.4.4.5.5 LLaMA Models

LLaMA [i.21] is a collection of foundation models that range from 7 billion to 65 billion parameters. This model family was trained on trillions of tokens (i.e. pieces of a word that make sense to the LLM) using publicly available datasets from 20 popular languages. The LLaMA models are notable for being small but powerful (e.g. it has been used to generate text, solve mathematical theorems, and predict protein structures). Its size requires far less computing power and resources to validate and test new applications. LLaMA is primarily a research tool that requires prompt engineering (see clause 4.5) to interact with. The LLaMA models are open source, and use the Apache® 2.0 license.

LLaMA 2 [i.22] has a number of model sizes, including 7, 13 and 70 billion parameters. Meta claims the pre-trained models have been trained on a massive dataset that was 40 % larger than the one used for LLaMA. The context length has also been expanded to 4 096 tokens, double the context length of LLaMA. LLaMA 2 uses grouped-query attention, which allows key and value projections to be shared across multiple heads in multi-head attention models, reducing memory costs associated with caching; this maintains performance while optimizing memory usage. The LLaMA 2 model goes through a series of supervised fine-tuning stages, followed by a cycle of RLHF to help provide a further degree of safety and responsibility. LLaMA 2 is open source for research and commercial use, but the license prohibits certain uses of the model, such as training other language models.

4.4.4.6 Sparse Transformers (informative)

Sparse transformers use a special type of attention, called sparse attention, to reduce computational and memory requirements. Sparse attention computes the attention weights for a subset of the input vectors. This subset can be chosen randomly or based on some importance measure. EXPHORMER [i.23] is a framework for building scalable graph transformers. Graph transformers extend the transformer architecture to graphs by using the attention mechanism to learn relationships between nodes in the graph. The attention mechanism works by computing a weighted sum of the node features, where the weights are determined by the similarity between the node features and a query vector.

Exphormer is a framework for building powerful and scalable graph transformers. It consists of a sparse attention mechanism based on two mechanisms: virtual global nodes and expander graphs. The virtual global nodes mechanism uses a set of virtual global nodes that are connected to all other nodes in the graph, enabling the attention mechanism to learn long-range dependencies between nodes in the graph, even if they are not directly connected. The expander graphs mechanism uses expander graphs (i.e. a type of graph that has good connectivity properties) to sparsify the attention matrix. This enables the attention mechanism to focus on the most important connections in the graph, while ignoring the less important connections. Overall, the sparse architecture uses fewer training parameters, is faster to train and has memory complexity linear in the size of the graph. These properties help scale to larger graphs. Exphormer is open source, and use the Apache® 2.0 license.

4.4.4.7 T5 Models (informative)

The Text-to-Text Transfer Transformer (T5) [i.24] uses the concept of transfer learning. Transfer learning typically is made up of two stages: training on a general dataset and fine-tuning over a more specific dataset (as in BERT, described in clause 4.4.4.4). T5 works by first converting the input task into a text-to-text format. T5 is then trained to generate the output text from the input text using masked language modelling. In masked language modelling, some of the words in the input text are masked out, and the model is trained to predict the missing words. The novelty of T5 is that it uses a unified framework for applying transfer learning to different problems.

FLAN-T5 [i.25] applies FLAN to the T5 model. FLAN stands for Finetuned LAnguage Net. It is a method for improving the zero-shot learning performance of natural language processing models by using instruction tuning (see clause 4.4.4.5.1). Interestingly, [i.25] proved that finetuning on only data that does not include Chain-Of-Thought (CoT) data degrades performance on CoT tasks by a significant amount. Therefore, FLAN-T5 jointly finetunes on both non-CoT and CoT data, resulting in much better CoT performance while maintaining performance on non-CoT tasks. A final benefit of instruction finetuning on CoT data both with and without exemplars is that the resulting model is able to perform CoT reasoning in a zero-shot setting. FLAN-T5 is multilingual (T5 is not), and comes in 5 versions (small, base, large, XL, and XXL versions that have 0,08, 0,25, 0,78, 3, and 11 billion parameters, respectively). [i.25] FLAN-T5 also uses chain-of-thought (see clause 4.5.2.4) to improve reasoning ability of the models. FLAN-T5 also uses RLHF in some of its datasets to fine-tune FLAN-T5 using RLHF (T5 did not use RLHF). It, like T5, is open source, and uses the Apache® 2.0 license.

CodeT5 [i.32] is a bimodal transformer model trained on code and text. CodeT5 is built on T5-large, and has 70 million parameters. uses a identifier-aware pre-training objective to better leverage the identifiers and a bimodal dual generation task to learn a better alignment between natural language and programming language using code and its comments. CodeT5 supports both code understanding and generation tasks.

CodeT5+ [i.33] is an extension of CodeT5. It is a family of models ranging from 220 to 770 million parameters.in which component modules can be flexibly combined to suit a wide range of downstream code tasks by its mixture of pretraining objectives on both unimodal and bimodal multilingual code corpora. It uses instruction tuning (see clause 4.6) for scaling.

Both CodeT5 and CodeT5+ can be used to translate natural language into code.

4.4.4.8 Mixture of Experts Model (informative)

A mixture of experts (MoE) model is a machine learning technique where multiple expert models are used to divide a problem space into homogeneous regions. It differs from ensemble techniques in that typically only one or a few expert models will be run, rather than combining results from all models.

[i.26] describes MoEBERT is an MoE model based that is based on BERT (see clause 4.4.4.4), which is a transformer-based language model. Each model in the MoE is itself a BERT model. The MoE architecture consists of two main components:

1) a set of expert models, each of which is a small BERT model that is trained for a specific domain or task; and

2) a gating network, which selects the appropriate expert model to use for a given input.

Each BERT model is trained on a different subset of the training data. The gating network selects the appropriate BERT model to use for a given input. Since each BERT model is trained on a different subset of the same data, each BERT model will have a different match (or level of expertise). MOEBERT uses an "importance score" to measure the importance of each BERT model's contribution. Since this compresses the model and causes a performance drop, the paper adapts a layer-wise task-specific distillation algorithm. MOEBERT is open source, and uses an Apache® 2.0 license.

4.4.5 Conclusions (normative)

Clause 5.8.3 of [1] defines the requirements for policy management, and [2] defines the ENI Extended Core and Extended Policy Information Models. One of the options for constructing policies is to formulate them using a DSL.

A DSL is a small domain-specific programming language, so it would be helpful if the model was trained using a code dataset. Otherwise, prompting or instruction tuning (see clauses 4.5 and 4.6, respectively) shall be used to ensure good translation capabilities. However, some of the transformer models reviewed have restrictive licenses, so this limits the described transformers to those that have open source licenses with the right to modify the model, such as the Apache® 2.0 license. A summary of the models that meet these two requirements follows:

1) T5 and FLAN-T5 were both trained on a massive dataset of text and code.
2) CodeT5 and CodeT5+ were specifically designed for coding, but can also translate natural language into code.
3) BERT was designed for language translation but cannot generate code.
4) CodeBERT can translate natural language into code since it was trained on a bimodal data set of natural language and code.

While any of the above choices may work, CodeT5+ is a more powerful model. However, it is a larger family of models and hence, may require more resources. In terms of their use for a DSL:

1) BERT, T5 and FLAN-T5 may all be used to translate natural language to the grammar of a DSL using appropriate prompts and fine-tuning or instruction tuning.
2) CodeT5+ is the most powerful coding model, but is not well suited for translating to a DSL. It should be preferred for pure coding tasks (e.g. the use of Java®, Python™, etc.).
3) CodeBERT may also be used for translating into code, but is not as powerful as CodeT5+.

4.5 Prompting for LLMs, Transformers, and Chatbots

4.5.1 Introduction (normative)

Prompting is one way of improving the performance of LLMs, transformers, and chatbots.

A chatbot is a computer program that simulates human conversation through text or voice interactions.

There are two main types of chatbots: rule-based and AI-powered. Rule-based chatbots are programmed with a set of rules that define how they should respond to certain prompts or questions. In contrast, AI-powered chatbots use machine learning to learn from and adapt to their users. This means that they can become more accurate and helpful over time. The present document will only consider AI-based chatbots.

Chatbots typically use LLMs to generate human-like responses to prompts that is both grammatically correct and semantically meaningful. When a user types in a question or statement, the chatbot shall use the LLM to generate a response that is relevant to the user's input. The chatbot may also use the LLM to translate text from one language to another, or to write different kinds of creative content. The present document will further restrict analysing chatbots that use an LLM-based architecture.

Significantly, prompting should be used for transformers and LLMs, and is not limited to being used with chatbots. However, for the rest of this clause, the term LLM will be used to include transformers and chatbots.

4.5.2 Prompting

4.5.2.1 Introduction (normative)

Prompting is a technique for adapting an LLM to a new task without retraining the model from scratch. Prompts are used as a means to interact with LLMs to accomplish a task. A prompt is an input that the model shall respond to. Prompts may include instructions, questions, or any other type of input, depending on the intended use of the model. Prompts may come from end-users or applications. While prompts can take non-textual forms (e.g. an image), the present document is restricted to textual prompts.

Prompts may also include specific constraints or requirements like tone, style, or even desired length of the response. For example, a prompt to write a letter to a friend can specify tone, word limit, and specific topics to include.

The quality and relevance of the response generated by the LLM is heavily dependent on the quality of the prompt. Hence, prompts play a critical role in customizing LLMs to ensure that the responses of the model meet the requirements of a custom use case.

The term *prompt engineering* refers to the process of designing a set of prompts to generate a specific output. Prompts play a critical role in obtaining optimal results from the model. Benefits of using prompts include improved accuracy, along with the model illustrating how it reach a conclusion or answer.

The following subclauses provide a non-exhaustive introduction to prompts that may be used in an ENI System.

4.5.2.2 Input-Output Prompts

4.5.2.2.1 Introduction (informative)

This type of prompt provides the LLM with an input and an output. The LLM is then tasked with generating text that is similar to the output.

4.5.2.2.2 Zero-Shot Prompts (informative)

Zero-shot means prompting the model without any example of expected behaviour from the model. The problem with zero-shot prompts is that without examples, the model possibly will not understand a key term or phrase in the prompt. In addition, when applications include complex conversations, decision making and more, a single prompt does not typically enable the LLM to understand the context of the prompt; this could lead to inaccurate or incorrect results.

4.5.2.2.3 Few-Shot Prompts (informative)

Few-shot prompts provide a small number of examples in the prompt to overcome the above difficulty. A few-shot prompt enables the model to learn without training. This is one way of engineering a prompt.

4.5.2.3 Self-Consistency Prompts (normative)

This type of prompt asks the LLM to generate text that is consistent with itself. This can be done by providing the LLM with a starting point and then asking it to continue the text in a way that is consistent with the starting point. Self-consistency prompts can be used to improve the coherence and accuracy of the text that the LLM generates, because the LLM shall ensure that any new text that is generated is consistent with previous text that it has already generated.

Self-consistency prompts may also be used to improve the creativity of the text that the LLM generates. This is because the LLM is forced to come up with new ideas that are consistent with the starting point. However, this could be challenging, because the LLM may not be able to come up with new ideas that are consistent with the starting point. This is one reason why LLMs are trained with very large datasets.

4.5.2.4 Chain-of-Thought Prompts (normative)

Chain-of-thought (CoT) prompting [i.3] refers to concatenating a series of prompts to achieve a task. The concatenation may be viewed as a set of intermediate natural language reasoning steps that lead to the final output. More specifically, CoT prompting breaks down multi-step problems into intermediate steps, allowing language models to tackle complex reasoning tasks that cannot be solved with standard prompting techniques. It improves the reasoning ability of LLMs by prompting them to generate a series of intermediate steps that lead to the final answer of a multi-step problem. CoT prompting enables LLMs to address more challenging problems, including advanced arithmetic, common sense, and symbolic reasoning.

CoT thought prompting is a method used to guide the responses of an AI model during its reasoning process in a step-by-step manner. It involves providing a series of prompts, each building on the previous one, that lead the model towards a desired output. In this process, the LLM starts by recognizing the user's input and understanding the context of the conversation. The LLM shall analyse the user's request and determine the best course of action to fulfil it. Then, the LLM should deliberate on the possible responses it can generate based on its internal knowledge and capabilities. The LLM shall then generate a response and review its response for any errors or inaccuracies before sending it to the user.

This process repeats with each new user input, allowing for dynamic and interactive conversations. It is important to understand that while this process is linear in description, in practice it can be quite complex with multiple factors influencing each step. CoT prompting leverages few-shot learning by inserting several examples of reasoning problems being solved within the prompt. Each example is paired with a chain of thought (or rationale) that augments the answer to a problem by textually explaining how the problem is solved step-by-step. Due to their few-shot learning capabilities, LLMs can learn to generate a rationale along with their answers by observing the exemplars within a CoT prompt, which can improve reasoning performance.

Similar to self-consistency prompts, CoT prompts can be used to improve the coherence, accuracy, and creativity of the text that the LLM generates. However, CoT prompts can also be challenging for the LLM because it may not be able to come up with new ideas that are consistent with the starting point. This is typically solved by training with a very large dataset.

An important variant of CoT is called *least-to-most prompting*. This approach extends CoT prompting to decompose a problem into a set of smaller steps that are solved one-at-a-time, where the output of each subproblem is used as an input for the next subproblem.

4.5.2.5 Tree-of-Thought Prompts (normative)

A Tree-of-Thought (ToT) prompt [i.27] generalizes the CoT model by proving the LLM with a starting point and then asking the LLM to explore different possible outcomes or conclusions over coherent units of text (thoughts) that serve as intermediate steps toward problem solving. ToT enables exploration over multiple paths of thought simultaneously. It does this by maintaining a tree of thoughts, where thoughts represent coherent language sequences that serve as intermediate steps toward solving a problem.

ToT provides a series of prompts, each branching out from the previous one, to lead the model towards a desired output. The LLM shall recognize the user's input and understand the context of the conversation. It shall then analyse the user's request and determine the best course of action to fulfil it by defining a tree consisting of multiple paths, each representing a different approach to generating a response. The LLM shall then evaluate each branch based on its potential to generate a satisfactory response, and select the most promising branch. It shall then generate a response based on that branch. The LLM should review its response for any errors or inaccuracies before sending it to the user.

This process repeats with each new user input, allowing for dynamic and interactive conversations. It is important to understand that while this process is linear in description, in practice it can be quite complex with multiple factors influencing each step.

ToT allows LMs to perform deliberate decision making by considering multiple different reasoning paths and self-evaluating choices to decide the next course of action, as well as looking ahead or backtracking when necessary to make global choices. This allows the LLM to generate text that is more comprehensive and coherent than text generated by traditional prompts. ToT requires more resources than other methods, but the modular flexibility of ToT allows users to customize such performance-cost trade-offs.

Similar to self-consistency prompts, ToT prompts Tree-of-thought prompts can be used to improve the coherence, creativity, and problem-solving skills of the LLM because it shall reason about the different alternatives it explores to solve the problem and connect the different branches of thought in a logical way. However, ToT prompts can also be challenging because it may not be able to discover all of the possible outcomes or conclusions.

4.5.2.6 Skills-in-Context Prompts (informative)

A Skill-in-Context (SiC) prompt [i.28] instructs an LLM how to compose basic skills to resolve more complex problems. The LLM is also prompted to provide examples of how the skills can be used to solve the problem. This is often called compositional generalization, which enables an LLM to solve problems that are harder than the ones they have seen. Like CoT and ToT prompts, SiC prompts provide a starting point and require the LLM to continue in a way that is consistent with the starting point. SiC prompts require both the skills and the compositional examples to be demonstrated within the same prompting context.

Ideally, SiC prompts teach the LLM to explicitly ground each of its reasoning steps on the (more elementary) skills provided in the prompts. Specifically, the SiC prompt is constructed from three main blocks. The first block contains the skills that LLMs may need to use in order to solve a more complex problem, which includes both descriptions of the skills and the instructions (with a few examples) on how to use them. The second part consists of a few exemplars that demonstrate how to explicitly compose these into a solution to a more complex problem. The last part is the problem to be solved.

4.5.2.7 Graph-of-Thought Prompts (normative)

Graph-of-Thought (GoT) [i.36] extends the CoT and ToT methods by conceptualizing the output of an LLM or transformer as a flexible graph structure. In this graph, individual units of information (referred to as "LLM thoughts") act as the nodes, while the connections between nodes symbolize the interdependencies among them. GoT expands on ToT by representing thoughts as directed acyclic graphs, which enables individual thoughts to be aggregated as well as used to loop over one thought. The key advantage of GoT over CoT and ToT is that it can combine multiple chains of reasoning (e.g. merge thoughts from different reasoning chains through aggregation nodes in the graph). This enables recombining useful knowledge from separate lines of reasoning into a better solution. This innovative approach facilitates the amalgamation of diverse LLM thoughts, yielding collaborative outcomes, encapsulating entire networks of thoughts, and even refining thoughts using iterative feedback loops.

Reasoning is done by traversing the graph and performing different operations on the nodes and edges. The operations can be used to combine thoughts, split thoughts, and generate new thoughts. In particular, GoT can represent more complex and nuanced relationships between thoughts. The LLM shall recognize the user's input and understand the context of the conversation. It shall then analyse the user's request and determine the best course of action to fulfil it by constructing a graph of thoughts, where each node represents a different approach to generating a response, and edges represent connections between these approaches. The LLM shall then evaluate each node in the graph based on its potential to generate a satisfactory response. The LLM may navigate through the graph, selecting nodes that lead towards a promising response. When complete, the LLM shall generate a response. It should review its response for any errors or inaccuracies before sending it to the user.

This process repeats with each new user input, allowing for dynamic and interactive conversations. It is important to understand that while this process is linear in description, in practice it can be quite complex with multiple factors influencing each step.

4.5.2.8 Common Procedures to Increase the Value of Prompting

4.5.2.8.1 Introduction (informative)

The following subclauses will briefly describe different ways to augment different types of prompting.

4.5.2.8.2 Active Learning (normative)

Active learning is a learning algorithm that can interactively query a user (or some other information source, commonly called an oracle) to label new data points with the desired outputs. CoT, ToT, GoT, and SiC prompting can each be combined with active learning to use the LLM itself to identify data that should be included in the training set.

For CoT, ToT and GoT prompting, the following steps can be used to combine either with active learning:

1) The LLM is given a prompt.
2) The LLM generates text that is consistent with the prompt.
3) The text is evaluated by a human or another information source.
4) If the text is accurate and coherent, then the prompt is added to the training set.
5) The process is repeated until the training set is complete.

For SiC prompting, the following steps should be used:

1) The LLM is given a prompt that specifies the skills that should be used.
2) The LLM generates text that uses the skills to solve a problem.

3) The text is evaluated by a human or another information source.

4) If the text is accurate and coherent, then the prompt is added to the training set.

5) The process is repeated until the training set is complete.

4.5.2.8.3 Information Retrieval (informative)

LLMs typically have a limited context window. One way to solve this is to use information retrieval mechanisms to provide LLMs with access to additional information. A recent trend is to use vector databases, as follows:

1) Chunking the text into smaller parts (e.g. paragraphs, sentences, etc.).

2) Producing an embedding (i.e. a vector that represents the meaning of the chunk) for each chunk of text.

3) Storing these embeddings in a vector database.

4) Performing vector similarity search (based on these embeddings) to find relevant chunks of text to include in a prompt.

An embedding is based on how the algorithm(s) in the LLM work. The vector is typically created by using a machine learning model to learn the relationships between the words in the vocabulary of the LLM. The model can then be used to generate vectors for new words or phrases. For example, the algorithm(s) may analyse text based on meaning, grammar, sentiment, topic, or other features.

A vector similarity search finds the most similar vectors to a given vector. There are many open source libraries for performing a vector similarity search, such as FAISS (Facebook AI Similarity Search) [i.29] and vector search engines (e.g. Weaviate, available at https://sourceforge.net/projects/weaviate.mirror/).

4.5.3 Prompt Tuning (informative)

Prompt-tuning is an efficient, low-cost way of adapting an AI foundation model to new downstream tasks without retraining the model and updating its weights. Prompt-tuning involves using a small trainable model before using the LLM. The small model is used to encode the text prompt and generate task-specific virtual tokens. These virtual tokens are pre-appended to the prompt and passed to the LLM. When the tuning process is complete, these virtual tokens are stored in a lookup table and used during inference, replacing the smaller model. The prompts can be extra words introduced by a human, or AI-generated information introduced into the model.

Prompt-tuning allows a massive model to be trained to process new tasks with limited available data. It also eliminates the need to update the model's billions (or trillions) of weights, or parameters. Prompt tuning has been shown to improve the performance of LLMs on a variety of tasks while reducing required training data. It also uses less resources than instruction tuning, which retrains the model. In addition, models that are prompt-tuned on different tasks can be saved, without the need for large amounts of memory.

4.5.4 Prompt Templates (normative)

A prompt template is a pre-defined procedure for generating prompts. Prompt templates may be used for a variety of purposes, including generating specific text formats, answering questions in an explicit manner, and translating languages. There are two types of prompt templates: natural language and structured. The former use natural language to convey instructions, while the latter use a specific non-varying format.

Structured prompt templates are typically made up of three different parts:

- context: the part of the template that provides the model with information about the topic of the text.
- prompt: the part of the template that asks the model to generate text.
- response: the part of the template that provides the model with an example of the type of text that it should generate.

A prompt template may include instructions, few-shot examples, and specific context and questions appropriate for a given task. Prompt templates may be used to generate the same prompt outline in multiple places, but with certain values changed. This can be useful for tasks where the same prompt structure can be used with different values to generate unique content for different applications.

If the LLM is not well-suited for expressing the ideas that the user wants to explore, then a prompt template can create a custom "mini-language" to teach the LLM. For example, the user may need to query the LLM about graph properties. A prompt template shall define the meaning of symbols or statements of its "graph mini-language" to the LLM. This is shown in the following example:

"From now on, whenever I type two identifiers separated by a "→", I am describing a graph. For example, "x→y" describes a part of a graph with nodes "x" and "y" connected by an unlabeled edge. If I type "x{l:a, d:b, w:c}→y", then I am adding the following properties to the edge: l is its label, d is its direction, and w is the weight of the edge. In this example, a shall be a text string, b shall be either 1 (for unidirectional) or 2 (bidirectional), and c shall be a positive integer."

The next set of examples focuses on using natural language to interact with the LLM itself to help create better prompts.

Here is a template that asks the LLM to always ask additional questions first to help it better answer a user's question:

"From now on, whenever I ask a question, follow these rules. First, generate up to five additional questions that would help you give a more accurate response. Second, assume that I know little about the topic that we are discussing and define any terms that are not general knowledge. Third. when I have answered all of your questions, combine the answers to each to produce the final answer to my original question."

Here is a template that asks the LLM to provide fact-checking:

"From now on, whenever I ask a question, follow these rules. First, generate a set of facts that your response depends on. Second, if any fact is incorrect, then your response is also incorrect, and tell me. Third, append the set of facts to your response."

Another example is based on the ToT approach, and ensures that the LLM always suggests alternative ways to accomplish the task. For example:

"From now on, whenever I ask a question about root cause analysis, if there are better approaches to accomplish the same goal, list those approaches, compare the pros and cons of each approach using the style that I used, and ask me for which approach I would like to use."

Here is a template that asks the LLM to suggest more refined prompts than entered.

"From now on, whenever I ask a question about security, suggest a better form of my question in the style I am using and ask me if I would like to use your version instead."

Here is a template that asks the LLM to explain the rationale behind its response when generating code.

"From now on, whenever you provide a response to a question from about code generation, follow the following rules. First, explain any assumptions that you made. Second, explain the reasoning to support your response. Third, support your response with any additional code samples. Fourth, define any potential limitations that your response has in order to provide a more accurate response."

Any of the above exemplary templates may be restricted to a particular context. For example, the following could be appended to any (except the first (graph) template, and the context may be as narrow or as broad as desired as long as the LLM understands the context:

"Only apply these instructions to the following domains: "security", "deploying a cloud application, and telemetry that contains faults."

4.5.5 Prompt Pipelines (informative)

In machine learning and AI, a pipeline is an end-to-end construct, which orchestrates a flow of events and data. The pipeline is kicked-off or initiated by a trigger; and based on certain events and parameters, a flow is followed which results in an output. A prompt pipeline is a sequence of steps that are used to generate a prompt (or set of prompts) for a language model. A prompt pipeline extends prompt templates by automatically injecting contextual reference data for each prompt. The pipeline is initiated by a user request, which is directed to a specific prompt template. The variables or placeholders in the pre-defined prompt template are populated with the question from the user and the knowledge to be searched from the knowledge store. This helps prevent the LLM from making up text that is incorrect or not real (often called a "hallucination"). It also ensures that the LLM uses current data. The composed prompt (or set of prompts) is then sent to the LLM and the LLM response is returned to the user.

A prompt pipeline is typically directed to a specific prompt template or set of prompt templates. The combination of prompt pipelines and prompt templates include a number of specific advantages:

1) LLM Prompts can be re-used, shared and fine-tuned.

2) Prompts can be used within context and is a measured way of controlling the generated content.

3) Templating of generative prompts allow for the programmability of prompts, storage and re-use.

4.5.6 Prompt Drift (informative)

Prompt drift occurs when a model's performance degrades due to changes in the prompts that it is given. For example, if a model is trained on a dataset of text prompts that are about telemetry, and then is given a prompt asking about energy efficiency, the model's performance can degrade because it is not trained on data about energy efficiency.

NOTE: Prompt drift is similar to concept drift [1]. Concept drift occurs when a model's performance degrades due to changes in the underlying distribution of the data that it is trained on. In contrast, prompt drift occurs when a model's performance degrades due to changes in the prompts it receives (as opposed to its underlying model).

Like concept drift, prompt drift consists of cascading inaccuracies. It can result in a divergence in the model output for a given input prompt when comparing different iterations of the same model. Prompt engineering, and especially CoT, ToT, SiC, and prompt templates are effective for fixing prompt drift, especially when one or more is combined with a prompt pipeline. In addition, monitoring and detecting drift among the features, predictions, and ground truth can help identify the root causes and consequences of the drift, allowing for appropriate actions to be taken.

Prompt drift is especially important when chaining together prompts. For example, if one input produces an unforeseen output, this can produce drift which is exacerbated in subsequent prompts. This can cause the LLM Response to be inaccurate or incorrect, since LLMs are non-deterministic.

4.5.7 The Use of Prompts in an ENI System (normative)

Prompt Engineering is rapidly evolving. Prompting is typically superior to fine-tuning in AI development because it enables much faster iteration cycles that have lower costs. Prompting eliminates virtually all the overhead, delays, and costs involved in data collection, model training, infrastructure, and licensing. This is because:

1) Prompts do not need to be trained, whereas fine-tuned models do. This in turn implies that prompts can be directly tested on production models. This means that:

   a) Prompting uses no additional resources for training, whereas fine-tuning typically uses extensive compute resources during training.

   b) Prompts can be dynamically changed at request time. In contrast, fine-tuned models cannot (i.e. they are static after being trained).

   c) A corollary of the above is that prompt variations can be quickly tried on a model to see what works best. In contrast, fine-tuning requires waiting for each model to finish being trained before it can be evaluated.

2) Prompts need just a few examples, while fine-tuning needs thousands of training data points. Acquiring training data itself can be costly and slow.

3) Prompts use general API access so no hosting or engineering overhead. Fine-tuned models need specialized hosting infrastructure.

4) Reduced licensing costs: Prompts use the standard licensed model pricing. Fine-tuning may require additional licensing fees.

It is also possible to automate prompt engineering. This is typically done using a dataset containing expected user input data and corresponding expected outputs, and a prompt template. These inputs are fed into an LLM, which then generates prompts for text generation. A prompt evaluation function evaluates each candidate prompt, and selects the prompt with the highest evaluation score. This approach can be augmented by using a human-in-the-loop approach to further increase accuracy and correctness. The advantage of this approach is that users do not have to be technical. Rather, the user relies on the ability of the LLM to understand natural language and generate prompts without having to provide formal specifications. The limits of this approach are:

1) the LLM is not given access to external tools; and

2) this is not (currently) used to generate prompts that chain multiple LLM queries together.

There is no single approach for providing the best or most efficient prompt for a given LLM or even an application using an LLM. This is because of the inherent diversity in how an LLM is trained combined with its architecture, as well as the different needs of different applications. However, the methods described in clauses 4.5.2.2 through 4.5.5 provide a description of the most promising mechanisms to use when performing prompt engineering.

Prompt engineering is preferred over fine-tuning when:

1) Responses shall be required quickly and easily.

2) The systems does not have a large amount of labelled data for the task.

3) Neither the time or resources required to fine-tune the model are available.

In general, an ENI LLM or transformer should use prompt engineering. However, there are some specialized situations in which fine-tuning is preferred. These include:

1) Use cases that require enriched knowledge using external data. Such use cases are tailor made for fine-tuning, and are difficult to be conveyed using prompts.

2) Use cases that require the best possible accuracy and performance for a given task.

3) Use cases in which the model needs to be deeply specialized. This can only be accomplished through extensive training with carefully curated data.

4) Use cases that have an inherently complex structure. For example, many structural patterns that can be identified using fine-tuning cannot be duplicated using prompts.

5) Fine-tuned models provide superior cost and latency benefits compared to prompts. In addition:

   a) Fine-tuning eliminates the extra tokens required by prompting, which is important if using a paid API.

   b) Fine-tuning allows creating smaller specialized models distilled from a large model, reducing hosting costs. Prompting always requires the large model.

6) Fine-tuning can teach models to generate output in a very rigid constrained structure that may be difficult to consistently enforce through prompting.

7) Some licensed models only permit fine-tuning but not prompting, so fine-tuning unlocks additional large pretrained models.

4.6 Instruction Tuning (normative)

Instruction tuning (also called fine-tuning) is used to align LLMs with user needs. Alignment is needed because most LLM model objectives (predicting the next token) are different from the user's desire for the LLM to follow its instructions. One of the central approaches in instruction using is incorporating user input; RLHF [i.30] is the most often used method to accomplish this and should be incorporated in ENI Systems. Its usual steps are:

1) Given a pretrained LLM, construct a set of prompts for with the LLM should produce aligned responses.

2) Use a team of trained humans to label correct responses.

3) Collect demonstration data, and train a supervised policy.

4) Collect comparison data between model outputs, and train a reward model based on outputs that human labellers prefer.

5) Optimize a policy against the reward model using a method such as proximal policy optimization.

6) Steps 4 and 5 can be iterated continuously; more comparison data is collected on the current best policy, which is used to train a new reward model and then a new policy.

Proximal Policy Optimization (PPO) is a family of model-free reinforcement learning algorithms. PPO algorithms are policy gradient methods, which means that they search the space of policies rather than assigning values to state-action pairs. PPO algorithms have some of the benefits of Trust Region Policy Optimization (TRPO) algorithms, but they are simpler to implement, more general, and have better sample complexity. It is done by using a different objective function.

Instruction tuning is a more efficient way to adapt an LLM to a new task than fine-tuning, which involves retraining the model on a dataset of examples from the new task. This is because instruction tuning does not require the LLM to learn a new set of weights, but only to learn how to interpret the instructions.

There are several variants of instruction tuning. InstructGPT [i.12] (described in clause 4.4.4.5.1) defines a process of adjusting the parameters of the LLM to improve its performance on a specific task without retraining the model from scratch. This can be done by manually adjusting the parameters or by using a machine learning algorithm. For example, an instruction tuning algorithm might be used to improve the performance of an LLM on a question answering task. FLAN-T5 [i.25] (described in clause 4.4.4.7) is based on the idea is that by using supervision to teach an LLM to perform tasks described via instructions, it will learn to follow instructions and do so even for unseen tasks. Both InstructGPT and FLAN-T5 use transfer learning.

Here are some of the key differences between prompts and instruction tuning:

1) Prompts are given to the LLM before it generates text, while instruction tuning is done after the LLM has been trained.

2) Prompts are typically used to provide the LLM with a starting point or a goal, while instruction tuning is used to adjust the parameters of the LLM to improve its performance on a specific task.

3) Prompts can be used to improve the performance of LLMs on a variety of tasks, while instruction tuning is typically used for tasks that are difficult to train LLMs for, such as question answering and summarization.

4.7 The Use of Knowledge Graphs to Improve Reasoning (normative)

NOTE: This topic is for the next phase of ENI. Hence, this clause is limited to a simple description of it.

Knowledge graphs explain factual knowledge using entities, relationships, and logical rules (e.g. by using formal logic). Knowledge graphs may enhance LLMs and/or transformers by providing external knowledge for inference and interpretability. However, they are not able to produce textual explanations of their knowledge. Hence, it is desirable to combine LLMs and/or transformers with knowledge graphs to provide more powerful reasoning.

This requires a suitable framework to support the integration of these two different technologies. The following are possible frameworks that may be used to integrate LLMs and/or transformers with knowledge graphs:

1) Chain-of-thought (see clause 4.5.2.4). CoT can be used to integrate LLMs and/or transformers with knowledge graphs by using the LLMs and/or transformers to generate the steps in the chain of thought and the knowledge graphs to provide the model with the knowledge it needs to generate the steps. This is a simple and linear method.

2) Tree-of-thought (see clause 4.5.2.5). ToT can be used to integrate LLMs and/or transformers with knowledge graphs by using a tree-like data structure to represent the knowledge in a Knowledge Graph. The ToT model is trained on a dataset of text and code that is annotated with Knowledge Graph entities. When generating an output, the ToT model traverses the tree and retrieves the relevant knowledge from the Knowledge Graph. This knowledge is then used to generate the final output. This is a more complex method that exploits hierarchy.

3) Graph-of-thought (see clause 4.5.2.7). GoT extends the CoT and ToT methods by conceptualizing the output of an LLM or transformer as a flexible graph structure. Since the knowledge in a GoT approach is represented as a directed acyclic graph, it provides the ability to loop over one thought or aggregate two thoughts or more together to form a better thought. The key advantages of GoT over CoT and ToT are that it can combine multiple chains of reasoning - GoT can merge thoughts from different chains through aggregation nodes in the graph. If this approach is selected, then it could take three forms:

    a) Develop a knowledge graph-enhanced LLM (or transformer), which incorporates the knowledge graph during the pre-training and inference phases of the LLM (or transformer),

    b) Develop an LLM (or transformer)-augmented knowledge graph, which leverages the LLM (or transformer) for different knowledge graph tasks, or

    c) Develop a framework in which an LLM (or transformer) and a knowledge graph play equal roles at arriving at a solution. This latter approach may be more complex, but has the advantage of using both in bi-directional reasoning driven by both data and knowledge. *This in particular may correspond to how telecom networks work in the best and most natural way.*

4) For any of the three approaches described in 3) above:

    a) Transformers and LLMs may be trained to link entities in text to entities in a knowledge graph. This may help them to better understand the meaning of text and to generate more informative responses.

    b) Transformers and LLMs may be used to check the accuracy of facts by comparing them to the knowledge graph. This may help them to avoid generating false or misleading information.

    c) Query expansion: Transformers and LLMs may be used to expand queries by adding relevant entities and relationships from a knowledge graph. This may help them to generate more comprehensive and informative responses.

    d) Transformers and LLMs may be used to perform knowledge-guided reasoning by using the knowledge graph to identify the entities and relationships that are relevant to a task. This may help them to improve their performance on a variety of tasks, such as question answering and summarization.

    e) Transformers and LLMs may be used to complete knowledge graphs by identifying and filling in missing entities and relationships. This may help to improve the coverage and accuracy of knowledge graphs.

4.8 Retrieval Augmented Generation (normative)

In-context learning is the ability of an LLM to learn information not through training, but by receiving new information in a carefully formatted prompt. Retrieval Augmented Generation (RAG) [i.35] is an architecture used to augment the functionality of an LLM by adding an information retrieval system that provides data for the LLM to use. This enables the developer to control the data used by an LLM when it formulates a response. RAG can be fine-tuned and its internal knowledge can be modified in an efficient manner and without needing the entire model to be retrained.

RAG models are typically trained in two stages. First, the LLM is trained on a large corpus of text and code. Then, the RAG model is trained to retrieve relevant documents from an external knowledge source and to incorporate the information from those documents into the LLM's outputs. Because RAG consists of two stages, RAG is typically implemented as a pipeline. The use of a pipeline should ensure that the pipeline is generating high-quality responses, as well as where the pipeline can be improved.

For example, one method for determining how the RAG pipeline is performing is to have the LLM generate a set of questions based on their training documents. Then, a subset of those questions is randomly picked as the base evaluation questions.

The RAG process comes in three key parts:

1) Retrieval: based on the prompt, the system shall retrieve relevant knowledge from a knowledge base.

2) Augmentation: the system shall combine the retrieved information with the initial prompt.

3) Generate: the system shall pass the augmented prompt to a large language model, generating the final output.

RAG shall use word vector embeddings to calculate the similarity between different documents and prompts. A word vector embedding takes individual words and translates them into a vector which represents its meaning. There are many algorithms that perform this task. One example is Word2Vec (https://github.com/dav/word2vec), a family of algorithms that use a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. RAG first generates a query vector from the prompt using a transformer encoder. The query vector is then used to search for relevant documents in a pre-computed database of document vectors. The top-k most similar documents are then retrieved and passed to the transformer decoder to generate a response. As an example, the word "king" could first be embedded, then the embedding for "man" could be subtracted, and the embedding for "woman" could be added; this would result in a vector who's nearest neighbour was the embedding for "queen".

RAG takes an input and shall retrieve a set of relevant documents from given a source. The documents are concatenated as context with the original input prompt and fed to the text generator which produces the final output. This makes RAG adaptive for situations where facts could evolve over time. This is very useful as the parametric knowledge of an LLM is static. RAG allows language models to bypass retraining, enabling access to the latest information for generating reliable outputs via retrieval-based generation.

NOTE: RAG is very sensitive to semantic nuances, including the use of singular vs. plural forms and complex semantics that involve related words that span long lengths. Ultimately, RAG depends on how well the underlying LLM is able to process semantics.

5 Transformer Architectural Requirements (normative)

5.1 Introduction

The following clauses define requirements for the ENI transformer architecture.

5.2 Transformer and LLM Usage in an ENI System

[TU1] Transformers and/or LLMs may be used in an ENI System.

[TU2] Transformers and/or LLMs may be used for constructing ENI Policies.

[TU3] Transformers and/or LLMs may be used for translating natural language inputs into ENI Policies.

[TU4] Transformers and/or LLMs may be used to provide services for a knowledge graph.

[TU5] A knowledge graph may provide services for Transformers and/or LLMs.

[TU6] Transformers and/or LLMs may be used in conjunction with a knowledge graph for providing solutions.

[TU7] Transformers based on T5 or FLAN-T5 should be used to generate EBNF specifications for intent policies.

[TU8] BERT-based transformers may be used to generate EBNF specifications for intent policies.

[TU9] Transformers based on CodeT5 or CodeT5+ should be used to generate code from EBNF specifications for intent policies.

5.3 Transformer and LLM Architectural Requirements

[TA1] Transformers and/or LLMs shall be trained on as large a corpus of bimodal data as possible.

[TA2] Transformers and/or LLMs should use any appropriate knowledge of the ENI System when creating, editing, deleting, or otherwise managing an ENI Policy.

[TA3] Transformers and/or LLMs should use knowledge from [2] for constructing an ENI Policy.

[TA4] Transformers and/or LLMs shall not be directly exposed to systems external to ENI.

[TA4.1] Transformers and/or LLMs shall reside within a Functional Block in the ENI System Architecture.

[TA4.2] Transformers and/or LLMs shall use ENI Internal Reference Points to communicate with other ENI internal Functional Blocks.

[TA5] Transformers and/or LLMs should use context information as defined in [1] for constructing an ENI Policy.

[TA5.1] The ENI System shall notify the Functional Block that contains Transformers and/or LLMs that it has updated context information.

[TA5.2] The ENI System shall make said context data available per the APIs of the selected Reference Point of the above communication.

[TA6] Transformers and/or LLMs should use context information as defined in [1] for constructing an ENI Policy.

[TA6.1] The ENI System shall notify the Functional Block that contains Transformers and/or LLMs that it has updated situational awareness information.

[TA6.2] The ENI System shall make said situational awareness information available per the APIs of the selected Reference Point of the above communication.

[TA7] Transformers and/or LLMs shall use the grammar of a DSL when translating an ENI Policy to a DSL.

[TA8] Transformers and/or LLMs should use RAG to increase the accuracy of generated text.

[TA9] Transformers and/or LLMs should use prompt engineering to increase the accuracy of generated text.

5.4 Transformer and LLM Reference Point Requirements

[TR1] Transformers and/or LLMs shall use appropriate Internal Reference Points as defined in [1].

[TR2] Transformers and/or LLMs shall use appropriate External Reference Points as defined in [1].

6 Transformer Architecture for ENI (normative)

6.1 Introduction

This clause defines a recommended approach for adding LLMs and/or transformers to the ENI System Architecture. It discusses the design principles and alternatives for using transformers and/or LLMs, additions to the ENI System Architecture [1] to accommodate transformers and/or LLMs, and how transformers and/or LLMs interact with other ENI System Functional Blocks.

From this point forward, the term "transformer" will be used to mean "transformers and/or LLMs".

6.2 Design Principles

6.2.1 Use Case

The following clauses discuss one exemplary use case: using the Transformer Management Functional Block to convert an intent in natural language to an ENI Intent Policy in Domain-Specific Language (DSL) format. There are many other use cases, which may be added to a future version of the present document.

6.2.2 Overview

Figure 6.2.2-1 shows a high-level overview of how transformers will function in ENI. It adds four Functional Blocks related to Transformer creation and management that are nested within the Policy Management Functional Block.

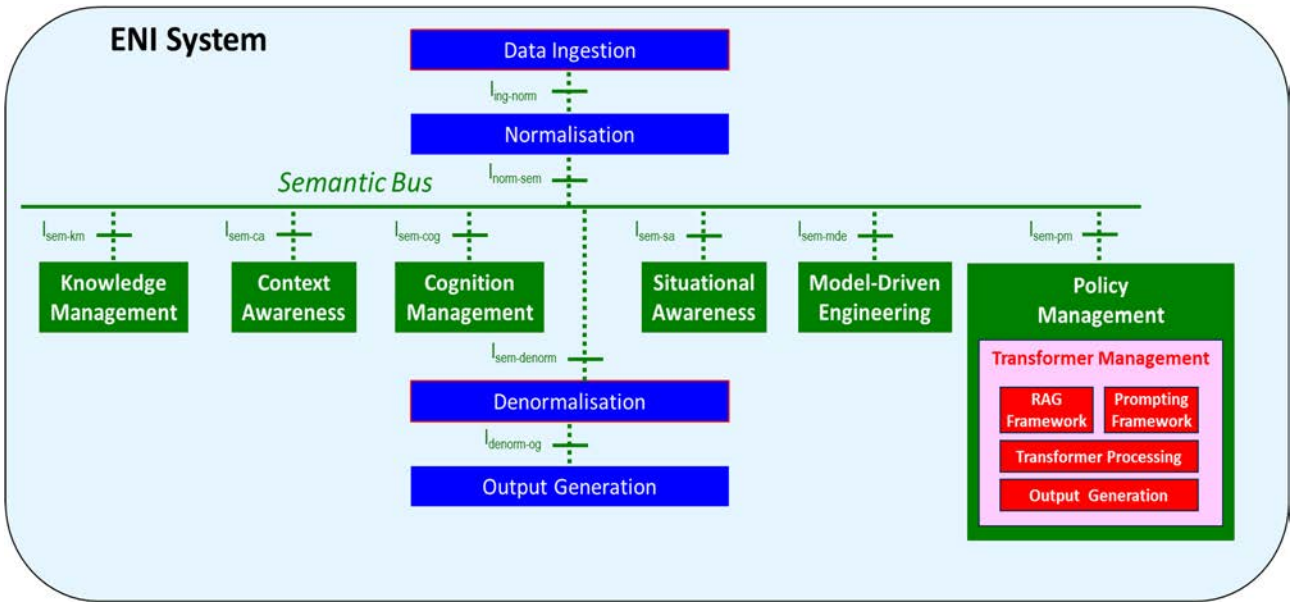

Figure 6.2.2-1: Policy Management Architecture (modified from [1])

NOTE: Documents for the RAG Framework may be located in the framework itself and/or in the Knowledge Repository (contained in the Knowledge Management Functional Block). The Prompting Framework may also contain documents to guide the generation of prompts in its framework and/or in the Knowledge Repository.

Conceptually, after an input intent policy is submitted to the ENI API Broker (not shown in Figure 6.2.2-1, but it is the intermediary between the Assisted System and the ENI System), the text shall be ingested and normalized and then sent over the Semantic Bus to the Policy Management Functional Block.

Referring to Figure 6.2.2-1, the Transformer Management Functional Block consists of a Transformer (middle block) that should be augmented by both a RAG Framework and a Prompting Framework. These functions are described in clauses 4.8 and 4.6, respectively. These two functions (RAG and prompting) represent two of the most efficient mechanisms to avoid retraining and enable the behaviour of a pre-trained model to be guided according to the needs of the application. As previously mentioned, these two functions are different. Hence, both of these functions should be used to augment the functionality of whatever transformer is chosen.

The Output Generation function provides a common interface for different post-processing operations. This use case shall use a DSL Generation function. Other use cases may use other output generation functions. This use case may combine the Output Generation and DSL Generation functions into a single interface.

At a high level, input text will be passed to the Policy Management Functional Block, where (for the purposes of this use case) it determines that the text is an intent policy. It then passes this information to the Transformer Management Functional Block, contained within it. The top three functions (RAG Framework, Prompting Framework, and Transformer Processing) collectively form what clauses 4 and 5 have described as a transformer. The fourth function (DSL generation) functions as a transpiler, converting natural language into an ENI DSL according to the grammar of that DSL. This output is then returned to the Policy Management Functional Block. Processing should then continue as described in clause 6.3.9.6 of [1].

6.2.3 Using Transformers vs. Traditional Compilers and Parsers

This architecture provides an alternative to traditional parsers and compilers. It should be used when the natural language is not (or only slightly) constrained, because the amount of ambiguity and problems with semantics require multiple processing layers to be added to a traditional parser or compiler.

More specifically, transformers offer a number of advantages over compilers and parsers, including:

1) Transformers are better at handling long-range dependencies in text. This is because transformers are able to attend to different parts of the input text when generating the output text (via self-attention). This is important for natural language translation because the meaning of a word or phrase can depend on the context in which it is used.

2) Transformers are better at handling non-sequential processing of text. Compilers and parsers process the words in a sentence, while transformers process the sentence as a whole.

3) Transformers are more data-efficient than compilers and parsers. This is because transformers are able to learn the rules of language from data, while compilers and parsers need to be explicitly programmed with these rules.

4) Transformers are highly parallelizable, which enables them to process larger amounts of text more quickly than parsers or compilers.

5) Transformers are more robust to noise in the input text. This is because transformers are able to learn the statistical distribution of language, which can be used to correct errors in the input text. This is especially true for structured text, such as DSLs.

6) Transformers are able to generate more fluent and natural-sounding translations than compilers and parsers. This is because transformers are able to learn the different styles of language that are used in different domains. In addition, parsers and compilers are designed to generate code, and typically do not understand the nuances of natural language.

6.3 Transformer Management Functional Block Architecture

6.3.1 Introduction

The Transformer Management Functional Block is located within the Policy Management Functional Block for two reasons:

1) The most common function of the Transformer Management Functional Block is to be used to create and edit ENI Policies.

2) External Policy Users (i.e. the End-User, an Application, the OSS, the BSS, and the Orchestrator) shall not have direct access to functionality provided by the Transformer Management Functional Block.

Since the Transformer Management Functional Block has no direct external access, no External Reference Points are needed. It shall receive and send information from and to the Policy Management Functional Block.

The Transformer Management Functional Block may require significant computing, memory, and/or other resources for its operation. This is one reason why it is designed as a Functional Block. Hence, it may use a set of Internal Reference Points, as detailed in clause 7 of the present document.

6.3.2 RAG Framework Functional Block

Retrieval Augmented Generation (RAG) uses a two-step process that retrieves relevant information and then generates responses based on that information. It is shown in Figure 6.3.2-1.

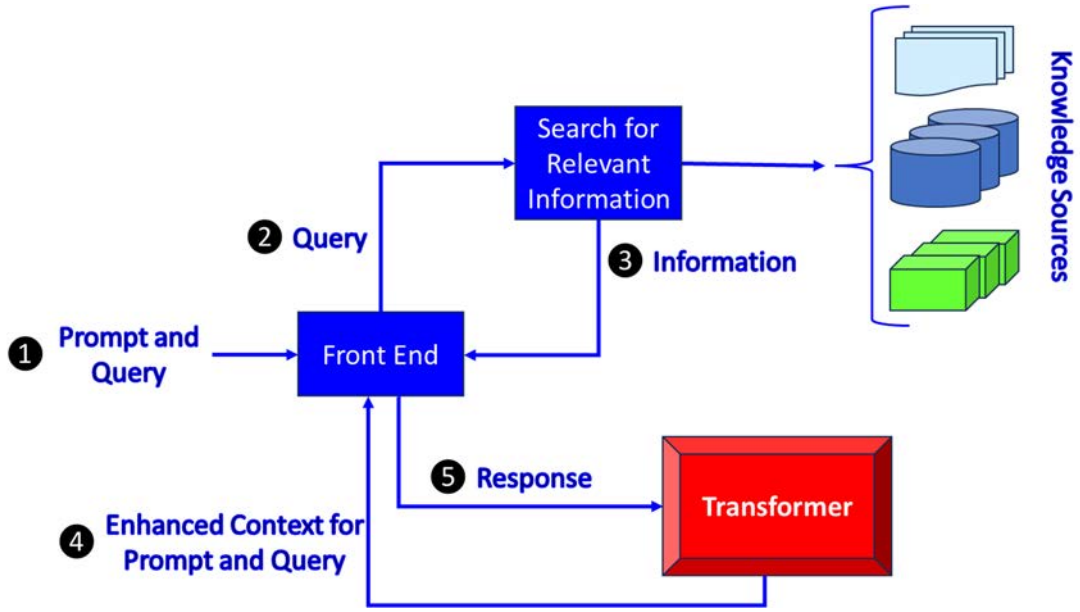

Figure 6.3.2-1: Conceptual Architecture of the RAG Framework

In the first step, the RAG model receives an input, such as a prompt. It then uses a retrieval system to search through a set of documents to find the ones that are most relevant to the input. This retrieval system uses a dense vector retrieval method, which allows the model to efficiently sift through the various knowledge sources and select the most pertinent documents to send back to the front end. The actual retrieval model used is based on the types of data sources and the needs of the application. Two examples are Best Match 25 (BM25, a probabilistic ranking function used by search engines to estimate the relevance of documents to a given search query) and DPR (Dense Passage Retrieval, a deep learning-based retrieval model that learns to represent text passages in a dense vector space by finding the passages with the most similar vectors).

Once the relevant documents have been retrieved, the RAG model moves on to the second step: response generation. The top retrieved results are concatenated into a context document that is provided to the transformer along with the original query. The transformer uses the retrieved information to help generate its response.

The advantages of the RAG framework include:

- Dynamicity. The transformer is supplied with relevant facts instead of having to rely on static training data.
- Context. The transformer is given a more accurate context to work with.
- Knowledge. Relevant external knowledge is provided to the transformer that is likely not contained in its training data, providing enhanced accuracy.
- Continuous Learning: As RAG models interact with more data, they become better at retrieving relevant documents and generating accurate responses.

- Auditability. RAG enables training outputs from retrieved information.

6.3.3 Prompting Framework Functional Block

Clauses 4.5.2.4, 4.5.2.5 and 4.5.2.7 described the chain-of-thought, tree-of-thought, and graph-of-thought prompting frameworks. They are ordered in terms of increasing complexity and accuracy. While all three of these prompting frameworks may be used, graph-of-thought should be used if its resource needs and complexity can be satisfied.

Figure 6.3.3-1 shows a functional architecture of the prompting framework. It assumes a single input modality (text), but could be easily expanded into multiple modalities by adding modality-specific encoders and increasing the cross-attention layering.

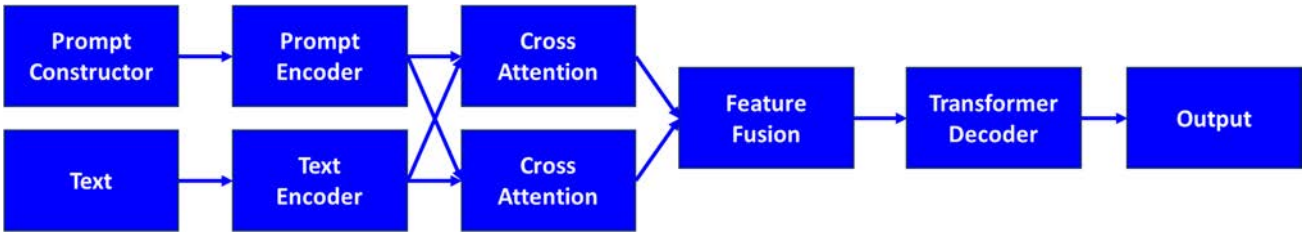

Figure 6.3.3-1: Functional Architecture of the Prompting Framework

The text is the input text, and the Prompt Constructor function may be any prompt engineering function, including CoT, ToT, or GoT. GoT is used from hereon to make the explanation more concrete.

The text is encoded using a transformer, and the output of the Prompt Encoder is encoded using an appropriate attention network (e.g. GoT uses a graph attention network). The cross-attention network aligns text with the prompt being used (e.g. text tokens are aligned with graph nodes). A feature fusion function is then used to fuse the text and GoT features and pass them into the Transformer decoder to predict the target thought decomposition. The feedback loop then takes the thought decomposition, concatenates them with the input text, and runs the same pipeline to produce the final results.

The Prompt Encoder is the key piece of this architecture, and performs the construction of the thought graph as follows:

1) Thought Unit Representation: As explained in clause 4.5.2.7, the information generated by an LLM is modelled as an arbitrary graph, where units of information ("transformer thoughts") are nodes and edges in the graph correspond to dependencies between these nodes. These dependencies may be used to represent any dependency relevant to the task at hand (e.g. the contextual relevance of the thoughts to each other or to the task being performed).

2) Graph-Enabled Transformations: This approach uses the inherent nature of the graph to apply graph-theoretic operations to the overall graph. This includes, but is not limited to, the following:

   a) Generation: a thought may be split into several refined thought paths, enabling each path to be pursued independently. For example, a complex problem may be split into multiple sub-problems.

   b) Aggregation: different thought paths in the graph may be merged to provide a single coherent and more robust thought.

   c) Merging: multiple thought paths are perceived as solutions to a problem that were previously generated. For example, when a complex problem has been split into multiple sub-problems, when each is solved, they can be merged into a single answer.

   d) Looping: this provides the ability to refine a thought by looping over a node or a set of nodes.

   e) Backtracking: this enables concluding that the current thought path is either no longer relevant or inferior in quality to another thought path, and hence, backtracking to pursue another thought path.

3) Non-Sequential Nature: By representing thought units as nodes and dependencies between them as edges, GoT captures the non-sequential nature of human thinking and allows for a more realistic modelling of thought processes.

Figure 6.3.3-2 compares CoT, ToT and GoT graphically.

This figure shows that:

1) All three approaches support ranking each intermediate thought.

2) All three approaches support abandoning a thought (for CoT, multiple CoTs are required to support this; each is processed linearly and compared at each step).

3) CoT processes intermediate thoughts in a linear sequence.

4) ToT represents intermediate thoughts as a tree (and not as a graph). As a tree, backtracking, along with simple operations (e.g. splitting and joining) are supported. However, each branch of the tree is independent of the others and cannot be combined.

5) GoT represents intermediate results as a graph. Hence, GoT supports the widest and most flexible range of operations that can be performed. For example, GoT supports both looping over one or more nodes to refine a thought as well as combining two or more paths to form a more powerful path.

6) Complexity increases from left-to-right, with CoT being the simplest to implement and GoT being the most complex to implement.

7) Accuracy also increases from left-to-right, with CoT being the least accurate and GoT being the most accurate.

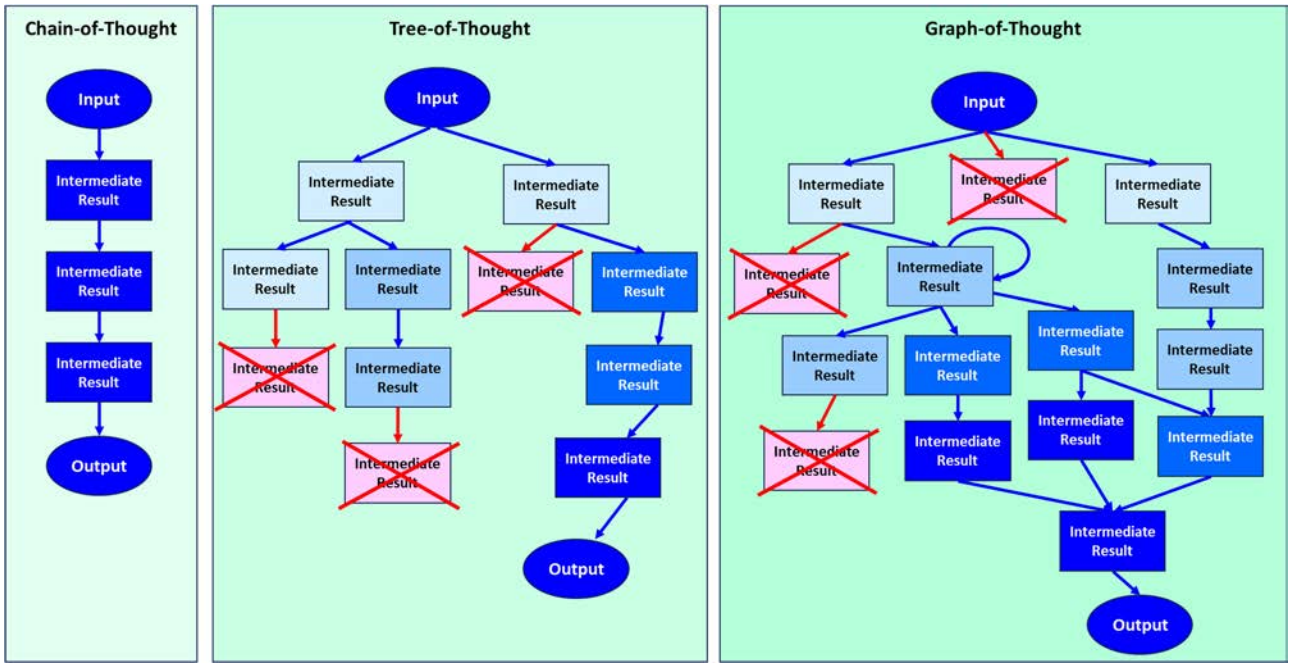

Figure 6.3.3-2: Visual Comparison of CoT, ToT, and GoT Prompting Method

6.3.4 Transformer Processing Functional Block

6.3.4.1 Overview

The landscape of transformers, both open and closed source, is rapidly changing. ETSI ENI should use an open source transformer to avoid any IPR issues.

6.3.4.2 Choice of Transformer to Use

As stated in clause 4.4.5, the initial (open source) transformer for ETSI ENI should be either BERT, T5, or FLAN-T5. More specifically:

1) T5 should be preferred over BERT because:

   a) T5 provides more flexibility.

b) T5 was defined specifically for text-to-text translation applications.

c) T5 is better for learning from structured data (since the target is a DSL, its grammar is structured).

2) FLAN-T5 may be preferred over T5 depending on the needs of the application:

a) FLAN-T5 was designed for zero- and few-shot learning. This means that:

i) FLAN-T5 models can be used to perform tasks without any prior training or with only a small amount of training data.

ii) FLAN-T5 could be trained on a small dataset of natural language sentences and their corresponding EBNF representations, and then be used to translate new natural language sentences to EBNF without any further training.

iii) Zero- and few-shot learning are preferable for translating between languages because they may be more efficient than traditional supervised learning methods, which require a large amount of labelled data to train. Zero- and few-shot learning models can learn to translate between languages with only a small amount of labelled data, or even without any labelled data at all.

b) FLAN-T5 is smaller (model sizes range from 6,7 to 137 billion parameters, compared to 11 to 540 billion parameters for T5).

c) FLAN-T5 has been shown to be more resistant to noise. This is important if there are errors in its training dataset.

3) T5 may be preferred over FLAN-T5 depending on the needs of the application:

a) T5 is more general purpose than FLAN-T5.

b) T5 has been shown to produce more accurate results than FLAN-T5.

c) T5 has been used more frequently than FLAN-T5.

6.3.4.3 Open Source Components Availability for the Transformer Processing FB

The DSL should be defined using EBNF. This is a rich structured representation of the grammatical rules specifying a language, and is portable across many platforms. The approach is to train T5 or FLAN-T5 as usual, and then finetune it with a corpus of natural language to EBNF translations. This would allow T5 to learn the relationship between the two languages and to generate EBNF representations from natural language sentences.

The HuggingFace Transformers library already contains support for using RAG with T5, and examples of using RAG with FLAN-T5 exist in open source. For CoT, ToT, and GoT, there are examples of T5 being used with each. However, there are only examples of FLAN-T5 being used with CoT.

It may be possible to generate the DSL directly without post-processing in the Output Generation Functional Block. However, in case the EBNF definition becomes complex, the RAG and Prompting Frameworks should be used to enable the translation to a DSL grammar. There are many open source frameworks for specifying an EBNF and generating code from the EBNF.

6.3.5 Output Generation Functional Block

Figure 6.3.5-1 shows the Output Generation Functional Block.

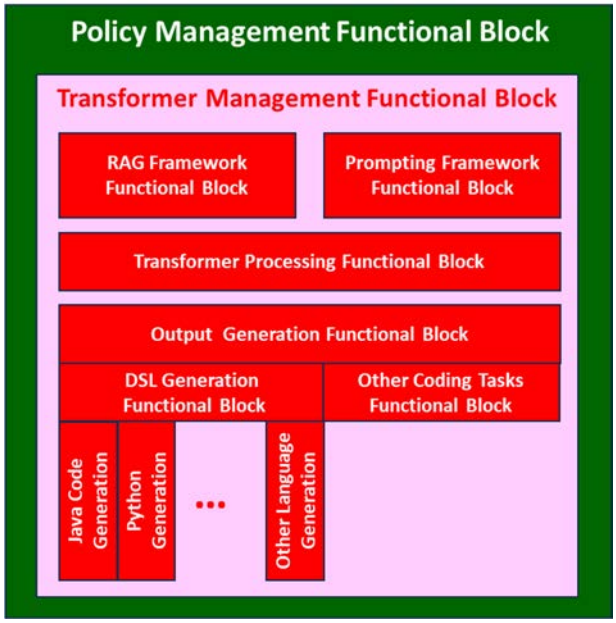

Figure 6.3.5-1: Output Generation Functional Block

The Output Generation Functional Block shall be used to generate code corresponding to the EBNF definition of the DSL as produced in the Transformer Processing Functional Block. It is designed as a modular set of hierarchical Functional Blocks. Two examples are shown. The present document is focussed on the DSL Generation Functional Block. As can be seen, it shall consist of one or more Functional Blocks that generate code for a particular programming language to implement the DSL.

The above architecture is modular and extensible, and can accommodate specific modules for different languages as well as different tasks.

Most machine learning and AI systems currently use Java® and Python™; hence, those are shown as examples. The actual programming language(s) will be defined in the next release of the present document.

As described in clause 4.4.5, the two top transformers for generating code and accompanying text, such as documentation, are CodeT5 and CodeT5+. They each support C, C#, Go, Java®, JavaScript, PHP, Python™, and Ruby. Code T5+ also supports C++.

The code for generating the DSL shall be based on the output of Transformer Processing Functional Block shown in Figure 6.2.2-1. This may include RAG and prompting functions.

There are two options for generating code:

1) Generate code directly from an EBNF specification using an open source parser generator.

2) Generate code directly by feeding the EBNF specification into CodeT5+.

The former typically produces a good start to coding but may produce some incomplete functions. ETSI ENI should use CodeT5+ (or another suitable transformer) instead of a parser generator because transformers are trained on large code datasets and have the ability to produce documentation for the code. In addition, CodeT5+ has special ability to leverage the code semantics conveyed from the developer-assigned identifiers. In addition, it has a rich API to communicate with.

6.4 Interaction with Other ENI System Functional Blocks

All interaction between the Transformer Management Functional Block and other ENI System Functional Blocks is done using the Policy Management Functional Block as a proxy. This provides indirect access for the Transformer Management Functional Block to all other functionality of the ENI System.

7 Transformer Internal Reference Points (normative)

7.1 Introduction

Since the Transformer Management Functional Block has no direct external access, no External Reference Points are needed. It shall receive and send information from and to the Policy Management Functional Block.

The Transformer Management Functional Block may require significant computing, memory, and/or other resources for its operation. Hence, it may use a set Internal Reference Points detailed in clause 6.3.

7.2 External Reference Points

External Reference Points connecting the Transformer Management Functional Block to external entities are not needed.

7.3 Internal Reference Points

The following Internal Reference Points are defined for communication between the Transformer Management Functional Block and the ENI System (I_{sem-pm} was already defined in [1], but is included here for completeness).

Table 7.3-1: Internal Reference Points between the ENI System and the Transformer Management Functional Block

Name	Brief Definition
I_{sem-pm}	Defines the data and information received by the Policy Management Functional Block from the Semantic Bus, as well as data and information that the Policy Management Functional Block publishes to the Semantic Bus. This is a bi-directional interface.
I_{pm-tm}	Defines the data and information received by the Transformer Management Functional Block from the Policy Management Functional Block, as well as data and information that the Policy Management Functional Block receives from the Transformer Management Functional Block. This is a bi-directional interface.
I_{tm-rag}	Defines the data and information received by the RAG Framework Functional Block from the Transformer Management Functional Block, as well as data and information that the RAG Framework Functional Block sends to the Transformer Management Functional Block. This is a bi-directional interface.
I_{tm-pf}	Defines the data and information received by the Prompting Framework Functional Block from the Transformer Management Functional Block, as well as data and information that the Prompting Framework Functional Block sends to the Transformer Management Functional Block. This is a bi-directional interface.
I_{tm-tp}	Defines the data and information received by the Transformer Processing Functional Block from the Transformer Management Functional Block, as well as data and information that the Transformer Processing Functional Block sends to the Transformer Management Functional Block. This is a bi-directional interface.
I_{tm-og}	Defines the data and information received by the Output Generation Functional Block from the Transformer Management Functional Block, as well as data and information that the Output Generation Functional Block sends to the Transformer Management Functional Block. This is a bi-directional interface.

8 Areas of Future Study (informative)

8.1 Open Issues for the present document

Void.

8.2 Issues for Future Study

From clause 4.7 (The Use of Knowledge Graphs to Improve Reasoning):

- A future version of the present document will define the interaction between knowledge graphs and transformers (and/or LLMs).

From clause 6.2.1 (Use Case):

- Additional use cases may be added to a future version of the present document.

History

Document history		
V4.1.1	March 2024	Publication