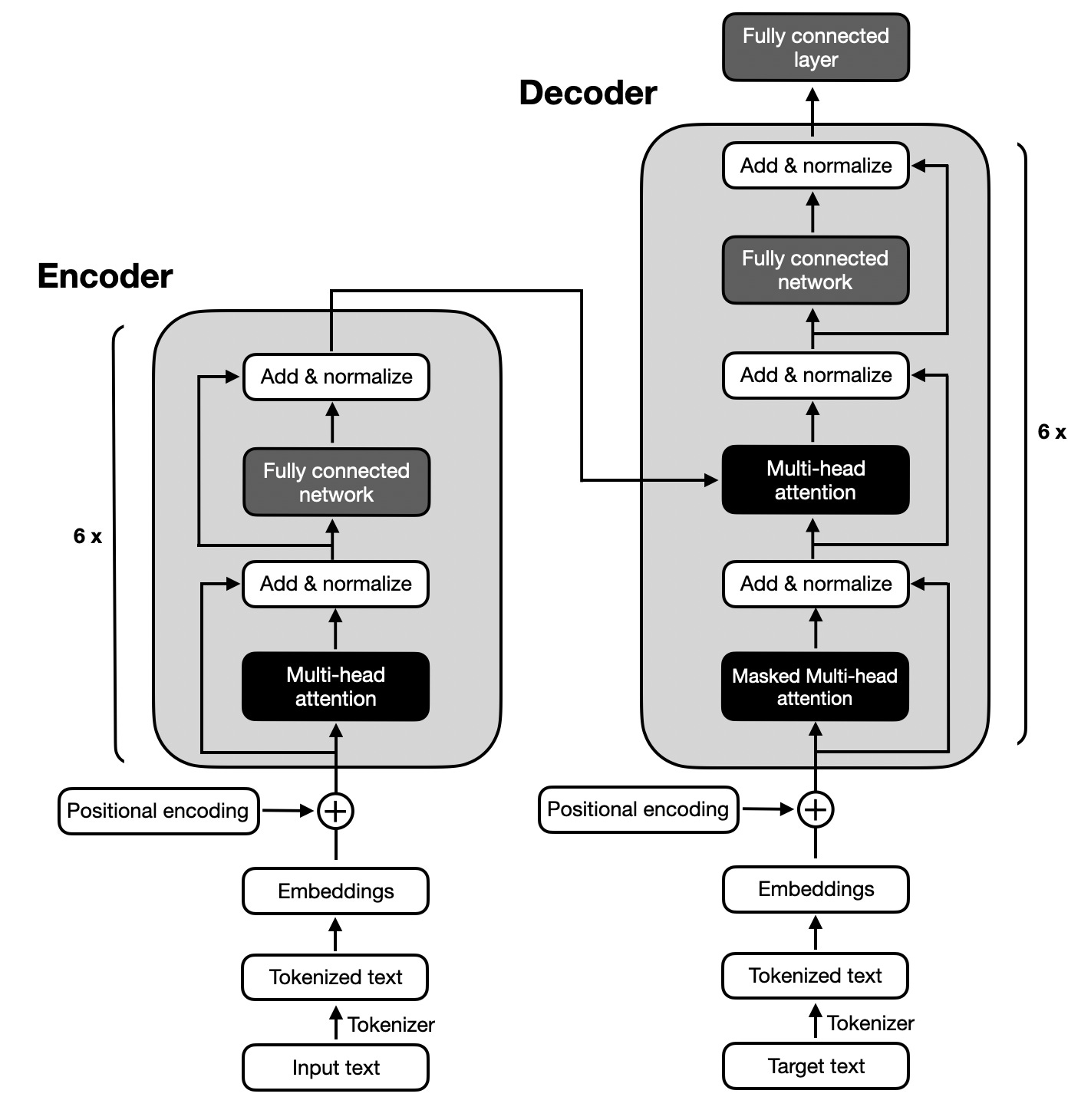
**Encoder - Decoder:**

**What is an Encoder-Decoder?**

Before knowing what encoder-decoder is, we will first see what they are used for? They are generally used to solve sequence to sequence problems, commonly known as seq2seq problems. These are sequence modelling tasks wherein both input and output are sequence. For example, a text summarization task wherein the input is a long text and output is also a text summarizing the input text. Thus, The model, often implemented with an [encoder-decoder architecture](https://www.google.com/search?rlz=1C1ONGR_enUS1145US1145&cs=0&sca_esv=05e16821a1d41d04&q=encoder-decoder+architecture&sa=X&ved=2ahUKEwjshriU75CNAxU1J0QIHSFEHIEQxccNegQIBRAB&mstk=AUtExfDW9kkDVmxFjrtPvVWjr-9W0NycE3Zi8ZiFJG_6J-IezAavI1STddwzGsu1mwzLB63MFvzooLve2HIbBv9R3qETw514Z2op1IdQMV6HAZ4W3IwaIyyXFmqwNmiaWMBFZacGSfuwQWtL3EyBpHDRoL_2aK0YxGy1tmevT2k5lj_A1_w&csui=3), learns to map an input sequence to a corresponding output sequence.

An encoder-decoder model is a type of neural network architecture that is explicitly designed for sequential data processing (or generation). The encoder-decoder model consists of two components, its encoder and decoder. The encoder receives the input sequence as input, compresses (and/or reshapes) that input sequence, and produces a fixed length representation. The decoder receives the fixed length representation as input, and decodes it to produce the output sequence. Because an encoder-decoder model can accomplish two distinct steps - an encode stage and a decode stage - thereby, allowing the input and output sequences to be of different length, encoder-decoder models are particularly useful for tasks where the size of the input sequence differs from the size of the output sequence, in areas including machine translation, text summarization, and image captioning.

**Workflow of encoder-decoder model (encoder-decoder Architecture):**



1. The tokenization stage:

In this phase, the input text (eg: a sentence of any language) is split into tokens. Tokens are generally the individual pieces that a text is broken down into, whether they be words, characters, or sub-words. Tokenization is done generally using subword units like Byte Pair Encoding (BPE) or SentencePiece.

1. The Embedding stage:

As machines can't understand any language directly and are very good with numbers, in the Embedding phase, each token is mapped to a fixed size dense vector (eg: 512-dimensional).

1. Adding Positional Encoding:

We know that in a sentence verbs are often followed by objects, adverbs, etc or adjectives before the noun they modify. But the machine does not know any sequence as such. Since the model processes tokens sequentially, it needs to understand the order of tokens in the input. Positional encodings are added to the embedding vectors to provide information about the position of each token in the sequence.

And this completes our preprocess. The final output of this stage is a sequence of embeddings enriched with position information — the actual input to the encoder block.

1. The Encoder stack:

The encoder consists of six identical layers that do the same thing but with different learned weights. This number of layers provides the model with depth in order to capture progressively more abstract and hierarchical representations of the input.

Each encoder layer contains:

* 1. Multi-head self Attention:

This part enables a token to attend to every token in a sequence — producing contextual representations. For example, in "List files modified today", the token "modified" may attend more heavily to "files" and "today" while building an understanding.

Multi-head attention means that this computation occurs in simultaneous attention heads, each learning different kinds of relations. For instance, one head might learn syntax while another learns temporal words, etc.

Mathematically, each head computes:

Where are projections of the input using learned weights

* 1. Add and Normalize:

The result of the multi-head attention is passed through a residual connection (the input is added back) and then normalized using LayerNorm. This stabilizes training and gradients.

* 1. Feed-Forward Network(FFN):

Each token is independently passed through a 2-layer feed-forward neural network:

This allows for more complicated transformations and non-linearity. The FFN is also followed by another Add & Normalize step. The final output of the encoder is a collection of context-enriched embeddings that represents every token and its relationships to the input. These will be used by the decoder later on.

1. The Decoder Stack:

The decoder is made up of 6 identical layers as well, but each layer includes just a little more complexity due to the fact that it also consumes the previous output tokens as well as the output of the encoder.

Each decoder layer has:

1. Masked Multi-Head Self-Attention:

This is identical to the encoder's self-attention, but it has a very important masking mechanism: tokens cannot attend to future tokens. For instance, when generating the third word, the decoder is only able to see the first two words. This prevents the encoder from making predictions simultaneously, which must happen in an orderly fashion during inference. The math is the same, except with a mask on top of the softmax to zero out attention to future positions.

1. Cross Attention (Encoder-Decoder Attention):

This is the point in the design where the decoder pays attention to the encoder outputs. The query comes from the previous layer of the decoder, whereas the key and value come from the last output of the encoder. This enables the decoder to look back at the input sequence and relate to the relevant tokens. So if the decoder is generating the bash command "ls .py", it might be attending to "Python files" in the input. The cross-attention component is at the heart of translation or mapping in the encoder-decoder structure.

1. Feed Forward Network:

Similar to the encoder, the output passes through a FFN and Add & Normalize. This process is repeated for each of the 6 decoder layers, producing gradually refined representations of the output sequence.

Final Output Layer:

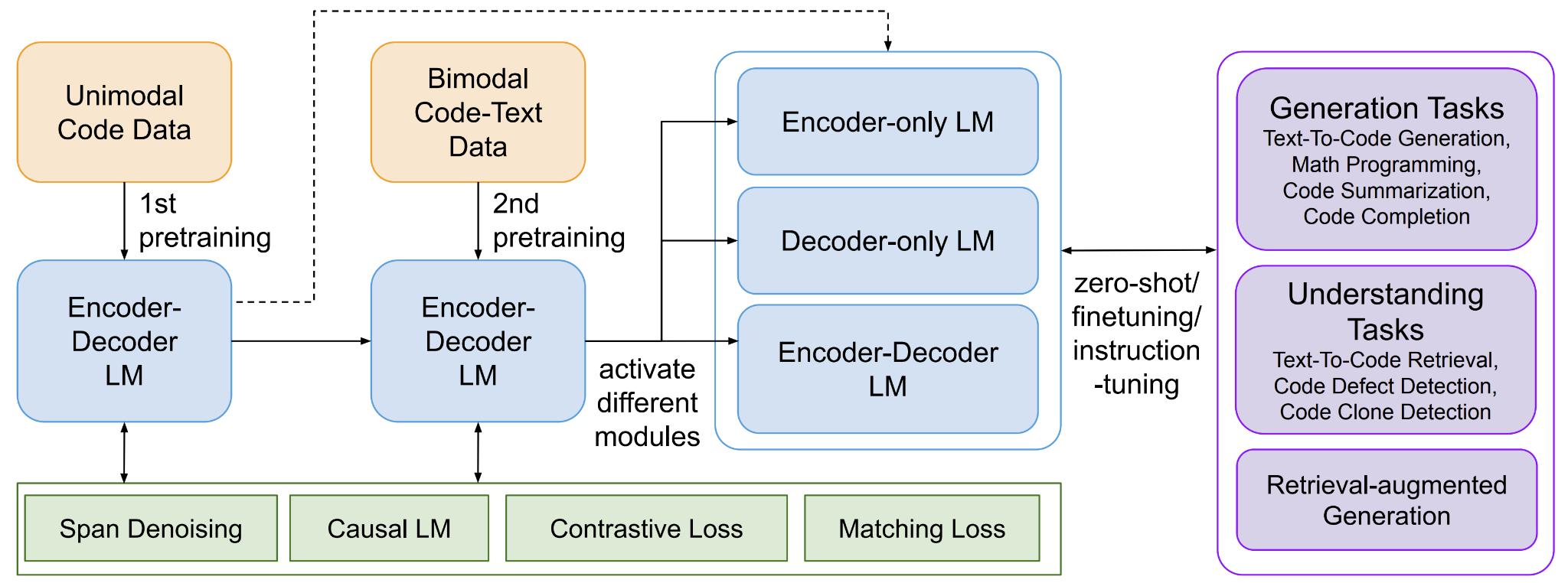
At the end of the last decoder layer, the result is then passed through a linear projection layer (fully connected) to project to vocabulary size. A softmax is then applied to produce a probability distribution over all next possible tokens and the decoder samples or simply takes the highest probability token and appends it to the output and proceeds until an <eos> token is generated or the maximum sequence length is reached.

**The CodeT5 Architecture:**

**A brief about what is CodeT5 model:**

CodeT5 is a research model from Salesforce Research that leverages transformers for code-based tasks. It is based on the T5 (Text-to-Text Transfer Transformer) architecture, which adapts to programming languages by interpreting both the input to the model and the output of the model as text. CodeT5 is trained using multi-task learning in multilingual programming languages (e.g., Python, Java, C, Ruby, etc), using procedures such as code generation, code summarization, translation from natural language to code, and code refinement. Additionally, by putting all tasks in a text-to-text framework, CodeT5 can utilize pretraining and fine-tuning processes similar to natural language models, but model the source code format and syntax as separate from natural language.

The next iteration is CodeT5+, which has benefits in supporting multilingual code, long input contexts, and enhanced controls over generation using span corruption, infilling, and prefix tuning. Available in multiple sizes (small, base, large), and implementation using popular benchmarks like CodeXGLUE and HumanEval, CodeT5+ is differentiated by its easy implementation using Hugging Face's `transformers` library and easy APIs to load, fine-tune, and evaluate model weights onto user-defined datasets which will make it useful for real-world software engineering and research.



**How Different is the CodeT5 model from the standard Encoder-Decoder Architecture?**

1. Text-to-Text Format for Code Tasks:

CodeT5 employs the T5-style text-to-text specification: everything (inputs, outputs) is treated as text - either code, comments, or natural language. Conventional encoder-decoder models are generally task specific, not uniformly text-to-text.

1. Pre Training with Code-Specific Objectives:

For instance, CodeT5 pretrains the model using denoising objectives that include span corruption, which masks and predicts parts of code or comments. Most models are pre trained using objectives like machine translation or language modeling without domain-specific modifications.

1. Multilingual Programming Language Support:

CodeT5 encompasses a large range of programming languages (Python, Java, C++, Ruby, etc.), due to training on diverse code corpora. Standard transformers are trained with human languages and need to be fine-tuned to code.

1. Code-Aware Pre Training Corpus:

CodeT5 is trained entirely on code-specific datasets (like CodeSearchNet, and Github repositories) allowing CodeT5 to learn syntactic and semantic structures in code. Traditional encoder-decoder models are trained on datasets that primarily consist of natural language (for example, texts from Wikipedia, books, etc.).

1. Multi-Task Learning Design:

CodeT5 has been designed for numerous code tasks such as code summary, generation, translation, refinement, and debugging, all within a single framework. The average encoder-decoder typically requires a separate fine-tuning pipeline for each of those tasks.

1. Span-Level Infilling:

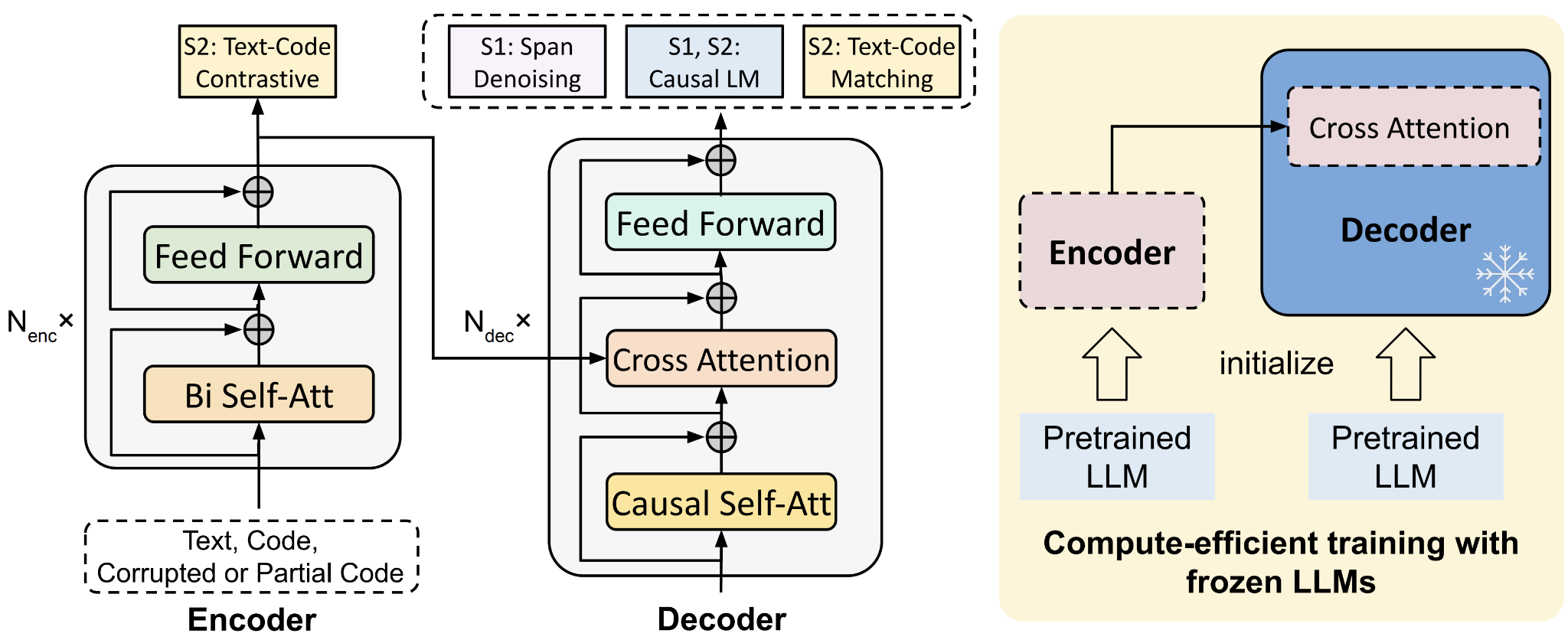
CodeT5 offers span infilling where the model learns to fill in gaps in code or comments; this is necessary for code repair and auto completion. Original transformers do not have span infilling, this is more of a BERT-style task.

1. Support for Custom Inputs (Prefix Tuning in CodeT5+):

The newer version of CodeT5+, supports prefix tuning, where users can control generation via prompts or structured prefixes - making it very useful for real-world IDE integrations. Standard models do not have this flexibility in the original offering.

1. Integration with Hugging Face for Easy Use:

CodeT5 is available with the Hugging Face transformers API, allowing plug-and-play fine-tuning on your own code data. To fine-tune a traditional transformer model for code-centric workflows, you'll likely require more effort to adjust, adapt, and change potential workflows to accommodate code data.



**Working of CodeT5 model:**

CodeT5 employs the Text-to-Text Transfer Transformer (T5) model architecture to work with programming languages and code-based tasks. CodeT5 considers all tasks, such as code generation, summarization, translation (NL ↔ code), or infilling as text-to-text problems. Thus, both inputs (natural language descriptions, partial code snippets) and outputs (functions, summaries, completed code) are in the form of text sequences. The model consists of an encoder to digest an input sequence and a decoder to output a sequence one token at a time, attending to both its own past outputs and the encoder's representations.

What makes CodeT5 particularly powerful for software engineering tasks is its code-aware pre-training strategy. During pre training, CodeT5 was trained using a denoising pre training objective (including span corruption and masked language modeling) across coronavirus code repositories (e.g., CodeSearchNet) to understand programming structure, syntax, and semantics, in order to facilitate a translation system between human language and code, as well as to generate syntactically well formed and semantically relevant code completions or fixes. Fine-tuning the model on downstream tasks (e.g., code summarisation & bug repairs) allowed the model to go through a specialisation phase as well. The more recent CodeT5+ version of CodeT5 has also incorporated smaller modifications.

**Why did we use the CodeT5 model for generating bash commands using natural language as input?**

Using the CodeT5 model to convert English-language prompts into matching Git or Bash commands is very promising because CodeT5 is built around the sequence transduction tasks of code. It is both an adaptation of and extension of the T5 model that enables it to utilize both natural language and programming languages in a joint text-to-text model. The model can treat the task as natural language → command line instruction, and utilizes its encoder to understand the intent and context, and its decoder to produce syntactically and semantically correct commands.

Additionally, while pretraining on a large corpora of code and natural language descriptions (e.g., GitHub repositories), CodeT5 implicitly learned relationships and patterns between descriptive phrases and executable code. This means, when fine-tuned on your custom dataset of English language prompts and Git commands, CodeT5 has become "domain specific" with respect to your syntax and idiosyncratic language. This lack of ambiguity is why you will have a high confidence that CodeT5 can correctly interpret your user-intent without hesitation; and not only generate valid bash commands, but can be used to automate virtually any terminal tasks, create a natural language interface or even aid developers with AI-enabled structure through tooling.

Links - <https://github.com/salesforce/CodeT5>