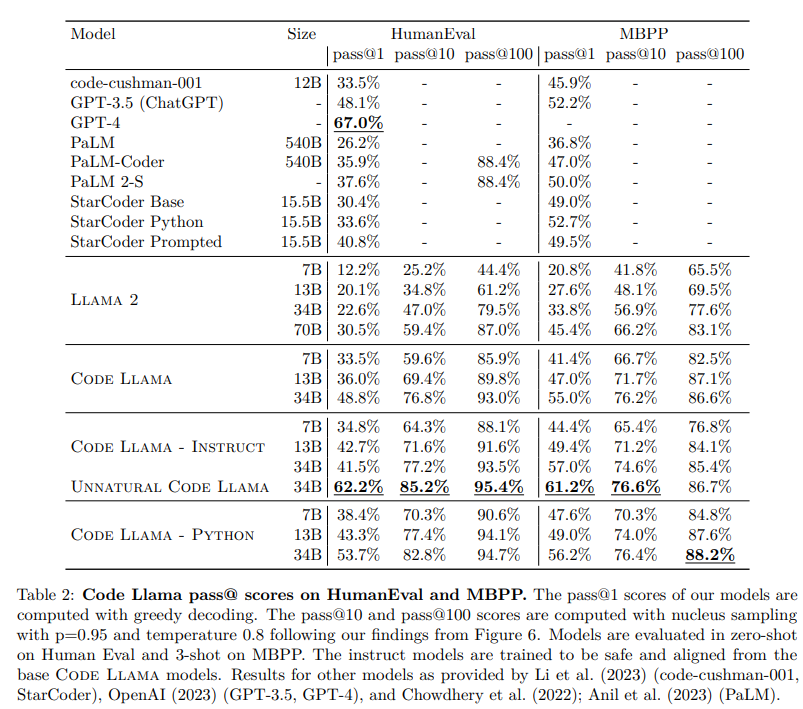
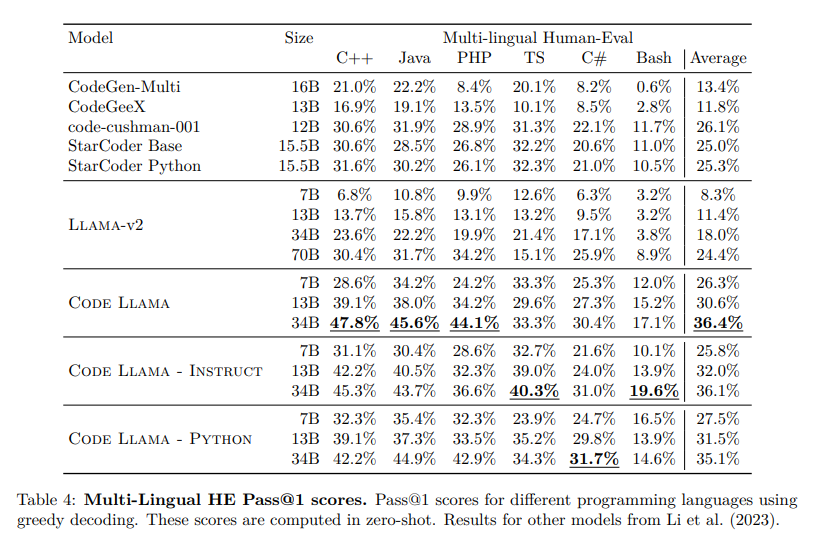
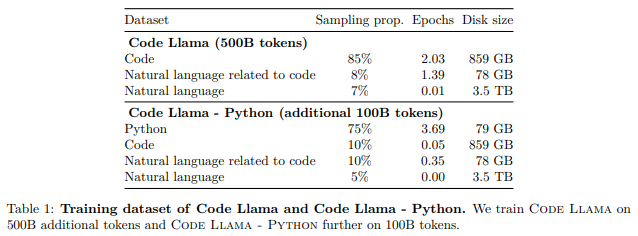
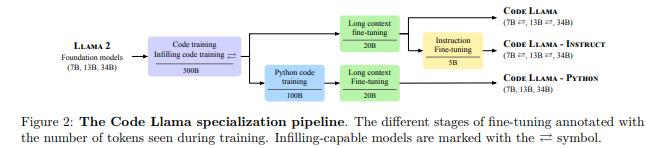
### **Code Llama Paper Review**

* **Results** [**https://arxiv.org/pdf/2308.12950.pdf**](https://arxiv.org/pdf/2308.12950.pdf)







**Important Paper Abstracts:**

* **Code-training from foundation models :** While most LLMs for code generation such as AlphaCode (Li et al., 2022), InCoder (Fried et al., 2023) or StarCoder (Li et al., 2023) are trained on code only, Codex (Chen et al., 2021) was fine-tuned from a general language model. We also start from a foundation model (Llama 2, Touvron et al., 2023b) pretrained on general-purpose text and code data. Our comparison (Section 3.4.1) shows that initializing our model with Llama 2 outperforms the same architecture trained on code only for a given budget
* **Unnatural model :** For comparison purposes, we also finetuned Code Llama - Python 34B on 15,000 unnatural instructions similarly to Honovich et al. (2023) using the same prompts as for the self-instruct dataset. We do not release this model, but we observe clear improvements on HumanEval and MBPP which are indicative of the improvements that can be reached with a small set of high-quality coding data
* **The value of model specialization :** Llama 2 was trained on 2T tokens, including only 80B tokens of code. Training on only 500B of extra tokens from a code-heavy dataset results in massive performance gains on both HumanEval and MBPP, to the point that Llama 2 70B is roughly equivalent to Code Llama 7B on Python coding benchmarks. Although Code Llama was trained on more than two epochs of our code dataset, which contains our entire Python dataset, training on 100B extra tokens of a Python-heavy data mix leads to significant gains on Python code generation benchmarks.
* **The value of self-instruct data** :.The Code Llama - Instruct models are trained to provide zero-shot instruction ability to Code Llama. In this further fine-tuning, where we somewhat distillate Llama 2-Chat, we focused not only on being more directly helpful (Figure 5c) but also sought to provide a safer model to use and deploy (Section 4). Following instruction and being overly safe can cost some points on evaluations. Further work is needed for LLMs to understand context and nuance in their instructions. The self-instruct data allows us to improve our scores on benchmarks such as HumanEval and MBPP. It also makes the training more reliable. With self-instruct, the model easily learns to follow the format requested for MBPP zero-shot while it sometimes fails without it
  + Please note: training a model with data generated using itself can lead to drift or mode collapse
  + Imagine training a mixture of Gaussians using data generated from the same mixture!
    - Ans: You expect even one mixture component’s overgeneration to make the model drift in the direction of that component: MODE COLLAPSE.
* **Additional modifications to LLM training and inference:** A number of works proposed to incorporate within the training objective structural knowledge of programs, with specialized objectives for code deobfuscation (Lachaux et al., 2021), contrastive learning through semantic-preserving code transformations (Jain et al., 2021), leveraging Abstract Syntax Trees to learn tree-aware positional encodings (Shiv & Quirk, 2019; Peng et al., 2021). A recent stream of work takes into account program execution or unit tests to filter, cluster, or improve the correctness of programs when few candidates must be submitted (Li et al., 2022; Chen et al., 2023a; Le et al., 2022; Zhang et al., 2023), or unit tests them within a reinforcement learning objective to enrich the training signal (Le et al., 2022; Liu et al., 2023a). We focused here on improving the base model rather than tweaking the inference scheme, since we believe this is where most of the long-term progress comes from; it is nonetheless an interesting direction to experiment with more elaborated inference schemes on top of Code Llama

* **Infilling :** Autoregressive training and fine-tuning of LLMs is suitable for prompt completion, but does not provide the capability to fill a missing portion of text while taking the full surrounding context into account. Our code-training for 7B and 13B Code Llama models features a multitask objective (Fried et al., 2023) consisting of both autoregressive and causal infilling prediction, enabling applications such as real-time completion in source code editors or docstring generation. Similarly to Bavarian et al. (2022); Li et al. (2023), our ablation study shows that infilling capabilities come at low cost in code generation performance for a given training compute budget (Section 3.2).
  + **Implementation Details:** We train infilling models following the concept of causal masking (Aghajanyan et al., 2022; Fried et al., 2023), where parts of a training sequence are moved to the end, and the reordered sequence is predicted autoregressively. We train the general-purpose 7B and 13B models with an infilling objective, following the 3 recommendations of Bavarian et al. (2022). More precisely, we split training documents at the character level into a prefix, a middle part and a suffix with the splitting locations sampled independently from a uniform distribution over the document length. We apply this transformation with a probability of 0.9 and to documents that are not cut across multiple model contexts only. We randomly format half of the splits in the prefix-suffix-middle (PSM) format and the other half in the compatible suffix-prefix-middle (SPM) format described in Bavarian et al. (2022, App. D). We extend Llama 2’s tokenizer with four special tokens that mark the beginning of the prefix, the middle part or the suffix, and the end of the infilling span. To limit the distribution shift between autoregressive and infilling training, we suppress the implicit leading space that SentencePiece tokenizers add upon encoding the middle part and the suffix (Kudo & Richardson, 2018). In SPM format, we concatenate the prefix and the middle part before encoding to tokens. Note that our model doesn’t encounter split subtokens in the SPM format while it does in the PSM format.
* **Long input contexts :** Unlocking repository-level reasoning for completion or synthesis – as opposed to function-level or file-level – requires prompting the model with much longer context than the 4,096 tokens supported by Llama 2. We propose an additional fine-tuning stage that extends the maximum context length from 4,096 tokens to 100,000 tokens by modifying the parameters of the RoPE positional embeddings (Su et al., 2021) used in Llama 2. LCFT comes at a cost for short sequences, and slightly decreases our scores on standard coding benchmarks such as HumanEval and MBPP. However, many real-world use cases are not captured by these benchmarks, and we believe that this cost is more than offset by the potential of handling long sequences for real downstream applications. Hence we opt to release all our Code Llama, Code Llama - Python and Code Llama - Instruct models with long-context capabilities.
  + **Implementation Details :** we propose a dedicated long context fine-tuning (LCFT) stage in which models are presented with sequences of 16,384 tokens, up from the 4,096 tokens used for Llama 2 and our initial code training stages. By limiting the training time spent on processing long sequences to a fine-tuning stage, we gain long-range capabilities without significantly increasing the cost of training our models. Our strategy is similar to the recently proposed fine-tuning by position interpolation (Chen et al., 2023b), and we confirm the importance of modifying the rotation frequencies of the rotary position embedding used in the Llama 2 foundation models (Su et al., 2021). However, instead of downscaling frequencies linearly as Chen et al. (2023b), we change the base period from which they are derived. Specifically, with rotary embeddings, the query and key vectors xn at position n are subject to a linear transformation Rd Θ,nxn, where Rd Θ,n is a block diagonal matrix with entries of the form shown below , and d denotes the embedding dimension. Rotation frequencies are computed as θi = θ −2i/d, and we increase the base period θ from 10,000 to 1,000,000 for fine-tuning. This increase allows for processing much larger sequences and reduces bias towards short-distance attention (see Appendix F.1 for further discussion). Our experiments confirm that Code Llama models are not only effective within the increased sequence length used during fine-tuning, but further show extrapolation capabilities and exhibit stable behavior on very long sequences of up to 100,000 tokens (Section 3.3
* **Instruction fine-tuning :** For end-users, the utility of LLMs is significantly improved by instruction fine-tuning (Ouyang et al., 2022; Wei et al., 2022; OpenAI, 2023; Touvron et al., 2023b), which also helps preventing unsafe, toxic or biased generations. Code Llama - Instruct variants are further fine-tuned on a mix of proprietary instruction data for improved safety and helpfulness, and a new machine-generated self-instruct dataset created by prompting Llama 2 for coding problems and Code Llama to generate associated unit tests and solutions. Our results show that Code Llama - Instruct significantly improves performance on various truthfulness, toxicity and bias benchmarks at moderate cost in terms of code generation performance
  + **Dataset Details**:
    - **Proprietary dataset :** Each example consists of a multi-turn dialogue between a user and an assistant. The dataset contains both Helpfulness and Safety data. This enables Code Llama to inherit Llama 2’s instruction following and safety properties
    - **Self-instruct :**  We construct the self-instruction dataset following the recipe below, resulting in ∼14,000 question-tests-solution triplets: 1. Generate 62,000 interview-style programming questions by prompting (Figure 9) Llama 2 70B. 2. De-duplicate the set of questions by removing exact duplicates, resulting in ∼52,000 questions. 3. For each of these questions: (a) Generate unit tests by prompting Code Llama 7B (Figure 10) (b) Generate ten Python solutions by prompting Code Llama 7B (Figure 11) (c) Run the unit tests on the ten solutions. Add the first solution that passes the tests (along with its corresponding question and tests) to the self-instruct dataset. We use Code Llama 7B to generate the tests and Python solutions, as we found it more efficient than generating fewer solutions per question with the 34B model for the same compute budget.
    - **Rehearsal :** In order to prevent the model from regressing on general coding and language understanding capabilities, Code Llama - Instruct is also trained with a small proportion of data from the code dataset (6%) and our natural language dataset (2%).
* **Training Details :** 
  + **Optimization :** Our optimizer is AdamW (Loshchilov & Hutter, 2019) with β1 and β2 values of 0.9 and 0.95. We use a cosine schedule with 1000 warm-up steps, and set the final learning rate to be 1/30th of the peak learning rate. We use a batch size of 4M tokens which are presented as sequences of 4,096 tokens each. Despite the standard practice of using lower learning rates in fine-tuning stages than in pre-training stages, we obtained best results when retaining the original learning rate of the Llama 2 base model. We carry these findings to the 13B and 34B models, and set their learning rates to 3e −4 and 1.5e −4 , respectively. For python fine-tuning, we set the initial learning rate to 1e −4 instead. For Code Llama - Instruct, we train with a batch size of 524,288 tokens and on approx. 5B tokens in total.
  + **Long context fine-tuning :** For long context fine-tuning (LCFT), we use a learning rate of 2e −5 , a sequence length of 16,384, and reset RoPE frequencies with a base value of θ = 106 . The batch size is set to 2M tokens for model sizes 7B and 13B and to 1M tokens for model size 34B, respectively. Training lasts for 10,000 gradient steps by default. We observed instabilities in downstream performance for certain configurations, and hence set the number of gradient steps to 11,000 for the 34B models and to 3,000 for Code Llama 7B.
* **Prompts :**

