

Final Project Field Guide

ML-Based Python Code Summarization

Outline:

1. Architecture Overview (DeepCom case study)
2. Practical Development Tips
3. References

What "done" looks like

- End-to-end training on Python code-summary pairs with reproducible configs and seeds
- Checkpoints + evaluation scripts yield stable BLEU/ROUGE/CE-loss numbers
- Inference summarizes new snippets without errors, timeouts, or CUDA OOM
- Report justifies choices and documents failure cases with evidence

Part 1: Architecture Overview

DeepCom: Deep Code Comment Generation (Hu et al., ICPC 2018)

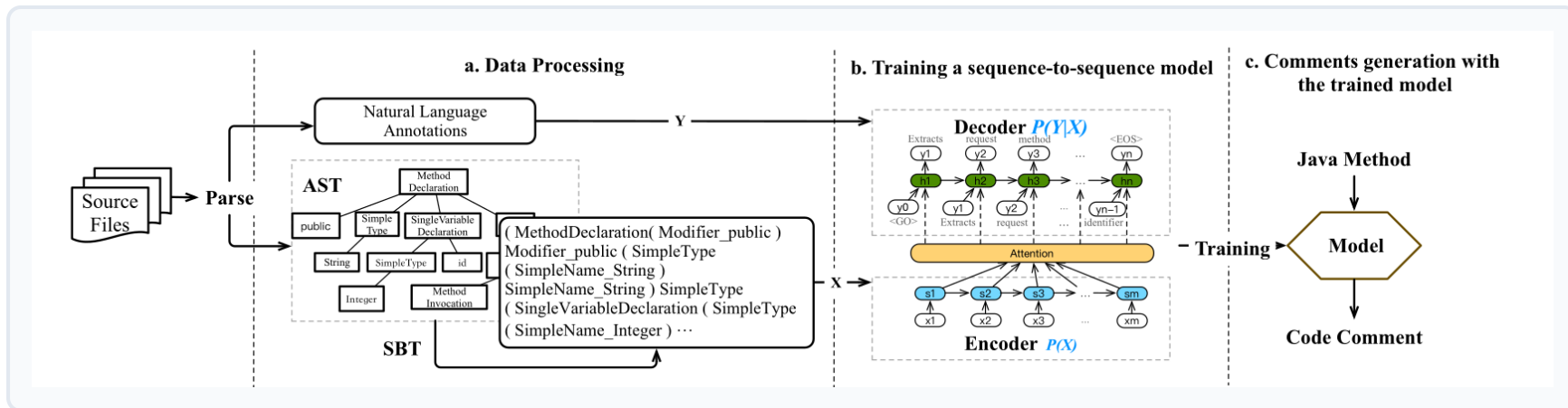
The Code Summarization Challenge

Why is generating code comments difficult?

- **Source code is structured:** Unlike natural language, code follows formal grammar with nested structures (classes, methods, control flow). Traditional NMT models struggle with this hierarchy.
- **Vocabulary explosion:** Code contains ~794k unique identifiers vs ~30k common words in NL. User-defined names (variables, functions) dominate the vocabulary.
- **Long-range dependencies:** A class used far from its import statement requires the model to capture distant relationships.

DeepCom's solution: Use Abstract Syntax Trees (AST) with Structure-Based Traversal (SBT) to capture code structure, combined with a Seq2Seq model with attention.

DeepCom Pipeline Overview



DeepCom Pipeline Overview

Three stages of DeepCom:

a) Data Processing

Parse source files into AST, convert to SBT sequences

b) Training

Feed AST sequences + comments into Seq2Seq model with attention

c) Inference

Generate comments for new Java methods using the trained model

Sequence-to-Sequence with Attention

Encoder

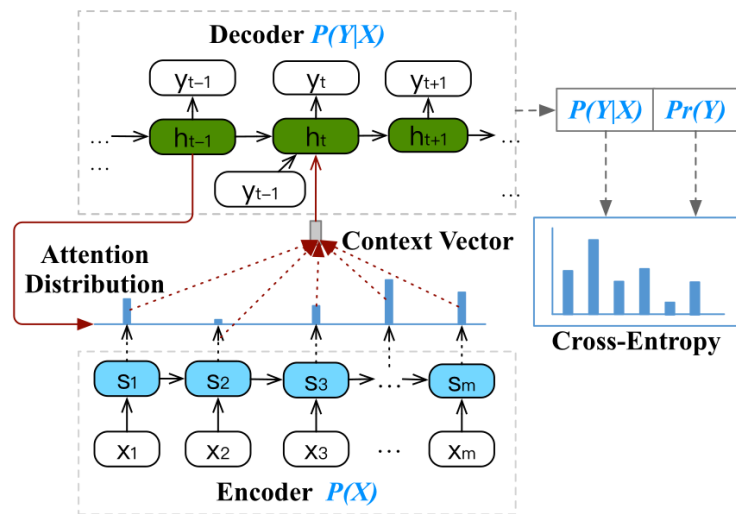
LSTM reads source sequence (x_1, \dots, x_m) ,
produces hidden states (s_1, \dots, s_m)

Attention

For each output word y_i , compute context vector:

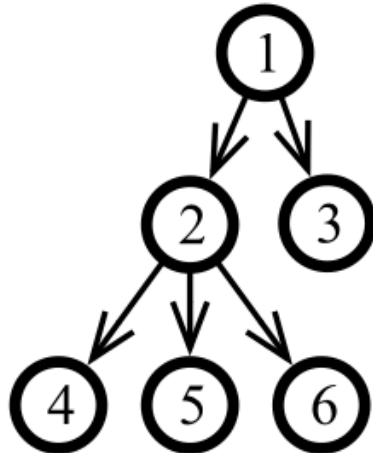
$$c_i = \sum_j \alpha_{ij} s_j$$

where α_{ij} weights how much each input position contributes.



Why AST? Structure-Based Traversal (SBT)

Tree structure



Sequence generated by SBT

(1(2(4)4(5)5(6)6)2(3)3)1

Why AST? Structure-Based Traversal (SBT)

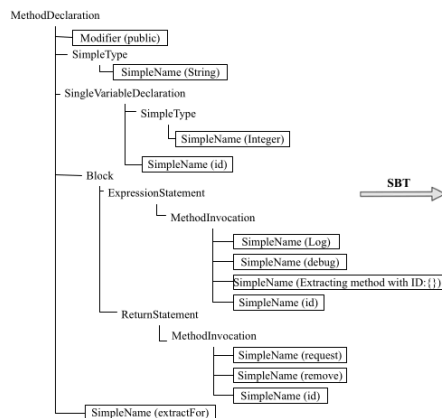
Problem: Plain source code as text loses structural information. Pre-order or post-order traversal of AST is *lossy* — the original tree cannot be reconstructed.

SBT Solution: Wrap each subtree in brackets with the node itself:

- Terminal nodes: `(node)node`
- Non-terminal nodes: `(node ... children ...)node`

Result: Tree `(1(2(4)4(5)5(6)6)2(3)3)1` — **lossless and unambiguous**. The original AST can be perfectly reconstructed.

Concrete Example: extractFor Method



SBT

```
( MethodDeclaration
  ( ( Modifier_public ) Modifier_public
    ( SimpleType
      ( ( SimpleName_String ) SimpleName_String
        ) SimpleType
      ( SingleVariableDeclaration
        ( SimpleType
          ( ( SimpleName_Integer ) SimpleName_Integer
            ) SimpleType
          ( ( SimpleName_id ) SimpleName_id
            ) SimpleVariableDeclaration
          ( Block
            ( ExpressionStatement
              ( MethodInvocation
                ( ( SimpleName_LOG ) SimpleName_LOG
                  ( ( SimpleName_debug ) SimpleName_debug
                    ( ( SimpleName_ExtractingmethodwithID_{} )
                      SimpleName_ExtractingmethodwithID_{}
                    ( ( SimpleName_id ) SimpleName_id
                      ) MethodInvocation
                ) ExpressionStatement
              ) ReturnStatement
            ( MethodInvocation
              ( ( SimpleName_request ) SimpleName_request
                ( ( SimpleName_remove ) SimpleName_remove
                  ( ( SimpleName_id ) SimpleName_id
                    ) MethodInvocation
              ) ReturnStatement
            ) Block
          ( ( SimpleName_extractFor ) SimpleName_extractFor
            ) MethodDeclaration
          ) MethodDeclaration
```

Concrete Example: extractFor Method

Java Method:

```
public String extractFor(Integer id){  
    LOG.debug("Extracting method with ID:{}", id);  
    return requests.remove(id);  
}
```

Left side: AST of the method. Non-terminal nodes (MethodDeclaration, Block) show structure; terminal nodes (SimpleName, SimpleType) hold values.

Right side: SBT sequence ready for the encoder — lossless representation of the tree structure.

Handling Out-of-Vocabulary Tokens

Challenge: Code has ~794k unique identifiers. Using `<UNK>` for all unknowns loses semantic information.

DeepCom's approach:

- AST nodes have *type* (e.g., `SimpleName`) and *value* (e.g., `"extractFor"`)
- For OOV tokens, replace with their *type* instead of `<UNK>`
- Example: `SimpleName_extractFor` → `SimpleName` (keeps the role, loses specific name)

Vocabulary: Keep top 30k tokens from AST sequences. Type-based replacement reduces unknowns while preserving structural semantics — much better than meaningless `<UNK>` tokens.

Part 2: Practical Development Tips

From Setup to Submission

Workflow Blueprint

- Start tiny: 500–2k pairs on CPU to validate data → train → eval → infer pipeline
- Automate entry points: `train.py`, `evaluate.py`, `summarize.py` with argparse defaults
- Config discipline: check in `config.yaml/json`; log git commit + config hash per run
- One change at a time: baseline → tokenizer tweak → model depth → scheduler; log each run

Anti-pattern: Jumping straight to full dataset + large model. Debug on tiny data first!

Data Handling Tips

- **Normalize early:** strip trailing spaces, standardize quotes/indent, lowercase summaries
- **Token caps:** code 256–512 tokens, summary 64–128; log dropped outliers
- **Split by file:** avoid data leakage; shuffle with a fixed seed
- **Unit-test preprocessing:** verify on 3–5 edge cases (decorators, nested defs, classes, long strings)
- **Cache tokenization:** preprocess once, save to disk, reload each session

Pro tip: Create a debug subset (1k examples) for rapid iteration. Stream batches for large datasets.

Model & Training Sanity

- **Baseline first:** encoder-decoder with embeddings 256–512, 4 heads, 2–4 layers
- **Stability tricks:** dropout 0.1–0.3
- **Monitor closely:** log loss every N steps; if spikes, lower LR, verify masks
- **Checkpointing:** early-stop on val loss/BLEU; keep best-loss and best-BLEU checkpoints separately

- **Optimizer:** AdamW (LR $\sim 3e-4$) • **Batching:** batch 16–32 • **Mixed precision:** `torch.cuda.amp` for memory + speed
- **Generation:** beam search (4–6), `max_length` 96–128, `length_penalty` 0.8–1.0

Compute & Colab Strategy

- Local smoke tests: 1–2 epochs on tiny subset to catch bugs before using GPU
- Colab optimization: `torch.backends.cudnn.benchmark = True` after fixing input shapes
- Mixed precision: `torch.cuda.amp` to fit larger batches; fallback is smaller batch, not larger model
- Memory hygiene: delete unused tensors; `torch.cuda.empty_cache()` between phases

- Mount Drive and save `state_dict`, tokenizer vocab, config every epoch
- Implement `--resume path/to/checkpoint.pt` restoring optimizer + scheduler
- Name checkpoints with epoch/step + loss + BLEU to avoid overwrites
- Plan runs in 30–60 min chunks with frequent saves — avoid 10h marathons

Compute Options (No GPU at Home)

- **Google Colab:** Free T4 GPU; Pro for longer sessions. Mount Drive for persistence.
- **Kaggle Notebooks:** Free T4/P100-class GPUs; persistent Datasets/Models for checkpoints
- **Paperspace Gradient:** Free GPU notebooks with time caps; persistent storage
- **CPU fallback:** Shrink model + sequence lengths; use for debugging/ablation

Key insight: Use GPU *only* for training. Download, clean, tokenize, test on CPU. Keep sessions busy!

Evaluation & Inspection

- Track metrics: CE/perplexity during training; BLEU + ROUGE on held-out set
 - Qualitative checks: Print 5–10 diverse summaries per epoch to spot overfitting or degenerate output
 - Length control: If outputs ramble, tune `min_length` , `max_length` , `num_beams` , `length_penalty`
 - Baseline comparison: Compare to trivial baseline (reuse docstring, first line) to justify gains
-
- Deterministic inference: Set seeds; use beam/greedy for grading, sampling only for exploration
 - Input validation: Guard against empty code, giant files, non-UTF8
 - CLI UX: Support `--input` , `--file` , stdin; optionally show top-K candidates

Reporting Playbook

- **Data section:** Source, filters, split strategy, token caps, samples dropped
- **Ablations:** Tokenizer, max length, depth/width, LR schedule — include training curves
- **Failure analysis:** Where summaries break (long control flow, heavy OOP, rare libs)
- **Repro recipe:** Hardware (CPU/GPU/Colab), exact commands, seeds, checkpoint paths

Golden rule: Someone should be able to reproduce your results from your report alone.

Debugging & Sanity Checks

- **Overfit on one example first:** If your model can't memorize 1 sample, something is broken
- **Check gradients:** Watch for NaN or zero gradients — indicates bugs in loss or architecture
- **Visualize attention:** Plot attention weights to see what the model "looks at" in the code
- **Verify padding:** Ensure loss ignores `<pad>` tokens; wrong masking = garbage learning

- **Common mistake:** Forgetting `model.eval()` during inference (dropout still active!)
- **Common mistake:** Teacher forcing mismatch — train with it, infer without it

Experiment Tracking

- Use tracking tools: Weights & Biases, MLflow, or TensorBoard for logging metrics/plots
- Log everything: hyperparameters, git commit hash, dataset version, random seeds
- Compare runs: Side-by-side comparison helps identify what actually improved performance
- Save artifacts: Best checkpoints, generated samples, confusion examples

Pro tip: A simple CSV/spreadsheet with (run_id, config, BLEU, notes) already helps a lot!

Time Management

- **Start early:** GPU training takes hours/days — don't wait until the last week
- **Have a fallback:** If fancy model fails, a working baseline is better than nothing
- **Iterate fast:** Quick experiments on small data → promising directions → scale up
- **Set milestones:** Week 1: data ready, Week 2: baseline training, Week 3: improvements...

- **Keep a lab notebook:** Log what you tried, what worked, what failed and why
- **Don't chase SOTA:** A well-documented simple model beats a broken complex one

Code Organization

```
project/
├── data/           # Data loading, preprocessing
├── models/         # Model architectures
├── configs/        # YAML/JSON config files
├── scripts/        # train.py, evaluate.py, summarize.py
├── notebooks/      # Exploration, visualization
├── checkpoints/    # Saved models (git-ignored)
└── requirements.txt
```

- **Modular code:** Easier to debug, test, and modify
- **requirements.txt:** Pin versions for reproducibility (`torch=2.0.1`)
- **README:** Setup instructions, how to run, expected results

Risk Checklist

Risk	Consequence
No preprocessing tests	Silent leakage / format bugs
No checkpoints	Lost progress after Colab disconnect
No token limits	OOM or throttling on Colab
No baseline comparison	Unclear if model actually improves
No qualitative samples	Metrics fine but outputs unusable
No config logging	Can't reproduce best run
No seed fixing	Non-deterministic results

Quick Start Commands

```
# Train with config
python train.py --config configs/base.yaml --max-code-len 512 --max-sum-len 96 --save-every 500

# Evaluate checkpoint
python evaluate.py --checkpoint checkpoints/best_bleu.pt --split val

# Inference on new code
python summarize.py --input "def foo(x): return x+1" --checkpoint checkpoints/best_bleu.pt
```

Checklist before running:

- ☐ Seeds locked in config
- ☐ Config file checked into git
- ☐ Drive/storage mounted for checkpoints
- ☐ `--resume` implemented for recovery

Part 3: References

Key Papers in Code Summarization

Foundational Work (2016–2019)

- Iyer et al., ACL 2016: "Summarizing source code using a neural attention model" — First neural attention for code summarization
- Allamanis et al., ICML 2016: "A convolutional attention network for extreme summarization" — Convolutional approach
- Hu et al., ICPC 2018: "Deep code comment generation" — DeepCom: AST + SBT + Seq2Seq ★
- Hu et al., IJCAI 2018: "Summarizing source code with transferred API knowledge"
- LeClair et al., ICSE 2019: "A neural model for generating natural language summaries of program subroutines"
- Chen et al., ICONIP 2019: "Neural code summarization with AST paths"

Transformer Era (2020–2022)

- Ahmad et al., ACL 2020: "A transformer-based approach for source code summarization"
- LeClair et al., ICPC 2020: "Improved code summarization via GNN"
- Choi et al., BigComp 2020: "Keyword memory networks for code summarization"
- Bui et al., SIGIR 2021: "Contrastive learning with semantic-preserving transformations"
- Liu et al., ICLR 2021: "Retrieval-augmented generation with hybrid GNN"
- Li J. et al., ASE 2021: "EditSum: A retrieve-and-edit framework"
- Guo J. et al., ACL 2022: "Hierarchical syntax with triplet position encodings"
- Ma Z. et al., ESEM 2022: "MMF3: Multi-modal fine-grained fusion"

Recent Advances (2023–2024)

- Gao S. et al., TOSEM 2023: "Code structure-guided transformer for code summarization"
- Ji et al., IJCNN 2023: "Semantic and structural transformer"
- Choi et al., IEEE Access 2023: "READSUM: Retrieval-augmented transformer"
- Li J. et al., SEKE 2023: "GraphPLBART: Graph + pretrained transformer"
- Guo Y. et al., SPE 2024: "Context-based transfer learning for low-resource summarization"
- Liang & Huang, IEEE Access 2024: "Transformer with relative position encodings"
- Hu X. et al., LREC/COLING 2024: "Reduce-redundancy-then-rerank"

Trend: From RNN/LSTM → Transformers → Pretrained models (CodeBERT, CodeT5) → Retrieval-augmented + multi-modal approaches

Good Luck!

Questions?

Remember:

- Start small and simple
- Iterate fast
- Save often