The Full Pipeline of Automatic Source Code Summarization (ASCS)

A Comprehensive Survey-Based Lecture

Press Space for next page →

Course Information

Master Course: Machine Learning for Software Analysis

Based on: Zhang et al., "A review of automatic source code summarization," Empirical Software Engineering, 2024

Authors of Survey:

- Xuejun Zhang
- Xia Hou
- Xiuming Qiao
- Wenfeng Song

Presenter: Fabio Pinelli, fabio.pinelli@imtlucca.it

Learning Outcomes

By the end of this lecture, you will be able to:

- **Understand** the complete ASCS pipeline and its four major stages
- **Compare** different source code modeling approaches (token, tree, graph, combination)
- **Analyze** various code summarization generation methods (template, IR, deep learning)
- **Evaluate** quality assessment metrics and their limitations
- **Identify** current challenges and future research directions

What is Code Summarization?

Code summarization are concise textual descriptions in natural language that elucidate the functionality and logic of source code.

Why is it important?

- Enhances program comprehension efficiency
- Reduces maintenance costs
- Mitigates errors during development
- Facilitates team collaboration
- Lowers learning curve for new developers

The Challenge

Real-world problem:

- Many projects lack detailed and meaningful code summarization
- Manual writing is costly and time-consuming
- Code complexity continues to escalate

Solution: Automatic Source Code Summarization (ASCS)

- Converts source code into natural language descriptions
- Automatically generates supplementary text for code
- Provides clear explanations of functionality, logic, and purpose

Historical Context

Timeline of ASCS Research:

- **2010:** First information retrieval-based method (Haiduc et al.)
- **2016:** First deep learning approach - CODE-NN (Iyer et al.)
- **2018-2021:** Rapid advancement with transformer models
- **2022-2024:** Large Language Models (LLMs) integration

Evolution:

- Early: Template-based and IR-based methods
- Modern: Deep learning dominates (Seq2Seq, Transformer, GNN, LLMs)

The ASCS Pipeline Overview

The overall process consists of **four major steps:**

- **Data Collection** - Gather source code and summaries
- **Source Code Modeling** - Extract and represent code features
- **Code Summarization Generation** - Generate natural language descriptions
- **Quality Evaluation** - Assess the generated summaries

Each step has its own **objectives** and **challenges**

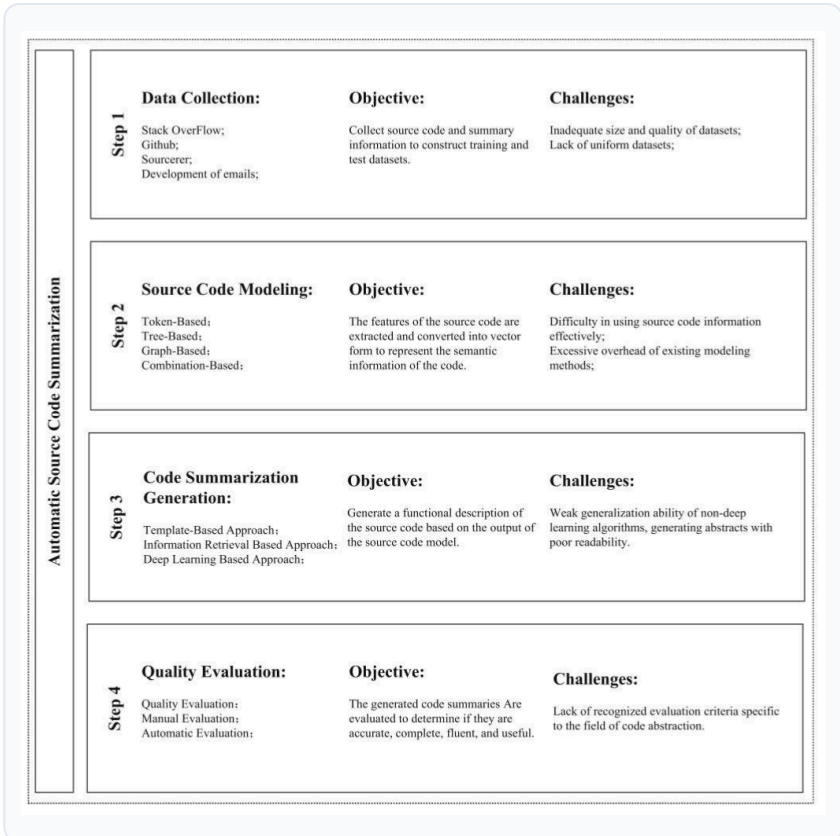
Pipeline

Step 1: Data Collection

- Sources: Stack Overflow, GitHub, Sourcerer
- Objective: Construct training/test datasets
- Challenges: Size, quality, uniformity

Step 2: Source Code Modeling

- Approaches: Token, Tree, Graph, Combination
- Objective: Extract semantic information
- Challenges: Information utilization, overhead



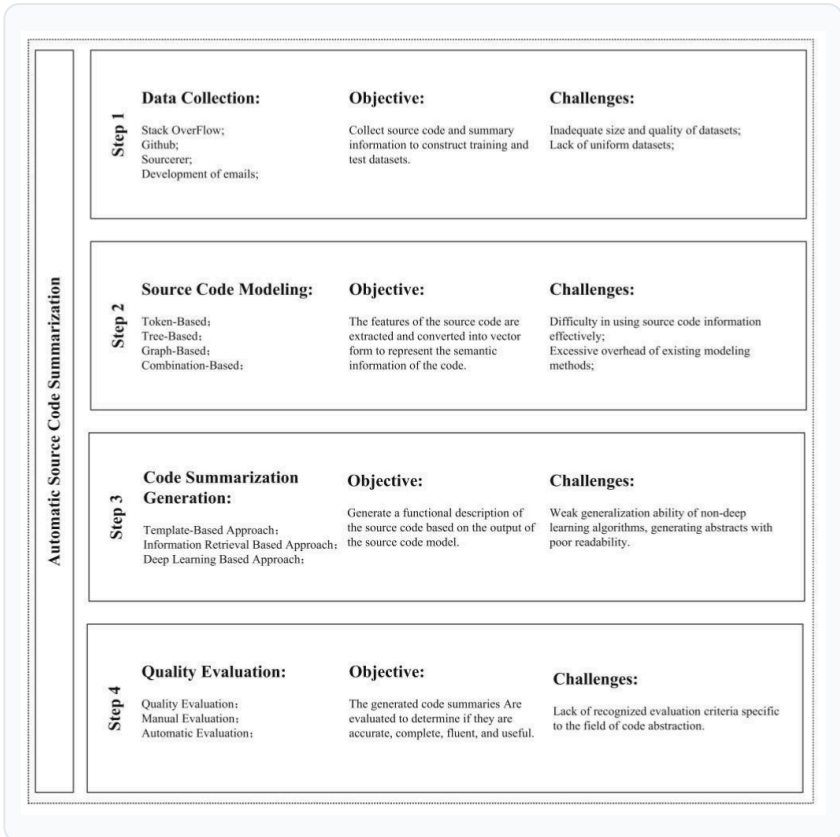
Pipeline (Continued)

Step 3: Code Summarization Generation

- Approaches: Template, IR, Deep Learning
- Objective: Generate functional descriptions
- Challenges: Generalization, readability

Step 4: Quality Evaluation

- Methods: Manual, Automatic
- Objective: Assess accuracy, completeness, fluency
- Challenges: Lack of recognized criteria



Part 1: Data Collection

Data Collection: Overview

Definition: The process of aggregating source code and pertinent summarization data from diverse sources to create datasets for training and testing.

Critical Importance:

- Success of code summarization methods is intrinsically linked to dataset quality
- Datasets mirror real-world demand for code summarization
- Provide insights into programming languages, domains, and annotation styles

Data Collection: Sources

Primary Data Sources:

- **Stack Overflow** - Programming Q&A with code snippets
- **GitHub** - Open-source repositories with documentation
- **Sourcerer** - Code search and analysis platform
- **Code Contests** - Competitive programming solutions

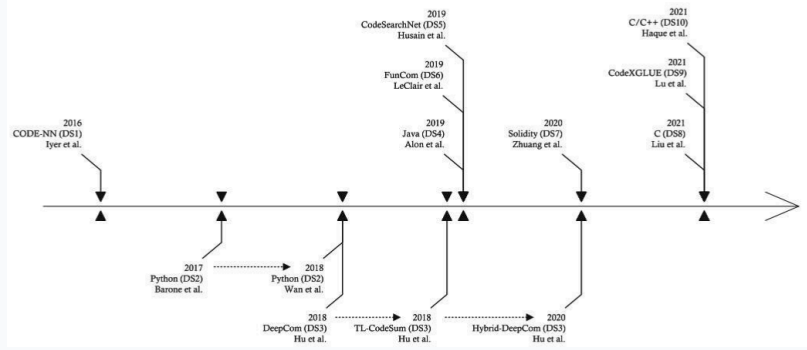
Data Types:

- Source code files
- Docstrings/comments
- Function/method descriptions

Dataset Evolution Timeline

Key Milestones:

- **2014:** First dataset (McBurney & McMillan)
- **2016:** CODE-NN dataset (Iyer et al.)
- **2018:** DeepCom dataset (Hu et al.)
- **2019:** CodeSearchNet, FunCom, Java datasets
- **2021:** CodeXGLUE, C/C++ datasets
- **2022-2024:** Quality improvement focus



Major Datasets: Overview

DS1 - CODE-NN (2016)

- Source: Stack Overflow
- Languages: C#, SQL
- Size: 66K C#, 32K SQL
- Granularity: Method

DS2 - Python Dataset (2017)

- Source: GitHub
- Language: Python
- Size: 161K pairs
- Granularity: Method

Major Datasets: DeepCom & CodeSearchNet

DS3 - DeepCom (2018)

- Source: GitHub
- Language: Java
- Size: 69K pairs
- Features: API sequences
- Granularity: Method

DS5 - CodeSearchNet (2019)

- Source: GitHub
- Languages: 6 (Go, Java, JS, PHP, Python, Ruby)
- Size: 2.1M+ pairs
- Granularity: Method

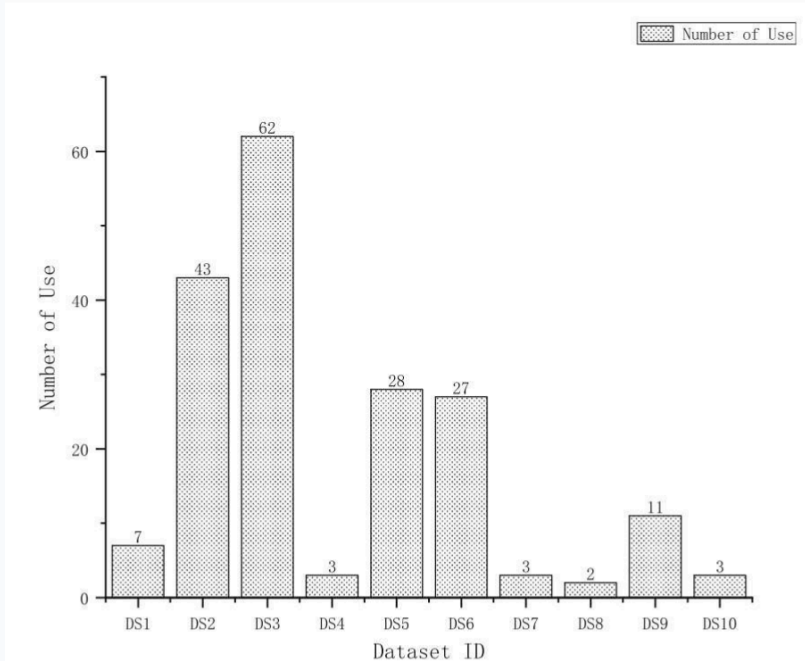
Dataset Usage Frequency

Most Popular Datasets:

1. **DS3** (DeepCom) - Java focus, high quality
2. **DS2** (Python) - Python language domain
3. **DS6** (FunCom) - 2M+ entries
4. **DS5 & DS9** - Multi-language support

Usage factors:

- Dataset quality
- Language coverage
- Dataset size
- Ease of access



Dataset Challenges: Scale

Scale Issues:

- Effective dataset should contain **hundreds of thousands** of samples
- Non-Java languages often have insufficient data
- Smaller datasets limit model performance
- May not cover enough syntactic/semantic variations

Impact:

- Restricts model generalization ability
- Limits practical application performance
- Affects model training effectiveness

Dataset Challenges: Quality

Quality Requirements:

- Comments accurately describe code functionality
- Adhere to consistent documentation standards
- Close semantic correlation between code and comments
- Rigorous data cleaning and filtering

Quality Improvement:

- CAT tool (2022) - 12 types of data noise detection
- Refined noise cleaning (2024) - "what", "how", "why" analysis

Dataset Challenges: Diversity

Diversity Needs:

- Multiple programming languages
- Different coding styles
- Various application domains
- Code snippets of different complexities

Future Directions:

- Expand to million+ samples
- Cover more programming languages
- Include diverse application domains
- Automated quality control tools

Part 2: Source Code Modeling

Source Code Modeling: Overview

Definition: The process of analyzing and modeling code to extract syntactic and semantic information, transforming it into vector form to represent code semantics.

Critical Role:

- Determines model's ability to understand code
- Directly impacts summarization quality
- Must handle code diversity and complexity

Source Code Modeling: Four Approaches

Classification:

- **Token-based** - Code as sequence of lexical units
- **Tree-based** - Abstract Syntax Trees (AST)
- **Graph-based** - Graph structures (CFG, DFG, PDG, CPG)
- **Combination-based** - Hybrid approaches

Each approach has unique strengths and limitations

2.1 Token-Based Modeling

Token-Based Modeling: Concept

Approach: Views code as a sequence of lexical units (tokens), similar to natural language text processing.

Process:

- Uses identifiers, variable names, method names as vocabulary
- Spaces, line breaks, indentation as separators
- Tokenizes code into sequence similar to natural language

Token-Based Example

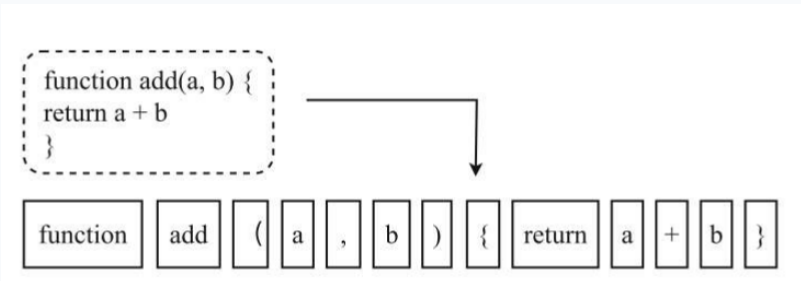
Figure 5: Token Representation

Source Code:

```
function add(a, b) {
  return a + b
}
```

Token Sequence:

```
function add ( a , b ) { return a + b }
```



Token-Based: Key Methods

Early Approaches:

- Haiduc et al. (2010b) - Text summarization techniques
- Wang et al. (2015) - Lexical annotation, keyword extraction

Modern Approaches:

- Hu et al. (2018b) - API knowledge integration
- Ahmad et al. (2020) - Transformer model for code tokens
- Hussain et al. (2020) - Tag type-based encoding

Token-Based: Advantages

Strengths:

- **Simple and effective** for lexical and syntactic analyses
- **Utilizes large-scale** open-source code for pre-training
- **Adaptable** to different programming languages and styles
- **Improves generalization** through pre-training and fine-tuning

Token-Based: Limitations

Weaknesses:

- **Relies on identifiers** - may be ambiguous or inconsistent
- **Surface information only** - ignores data types, control flow
- **Long dependencies** - difficult to handle nested loops
- **Context loss** - misses important structural details

2.2 Tree-Based Modeling

Tree-Based Modeling: Concept

Approach: Uses Abstract Syntax Trees (AST) to model code structure and semantic information.

AST Properties:

- Tree structure generated from syntactic rules
- Each code element (variable, operator, function) as a node
- Relationships modeled through edges
- Nodes have type and value (syntactic category and content)

Tree-Based Example

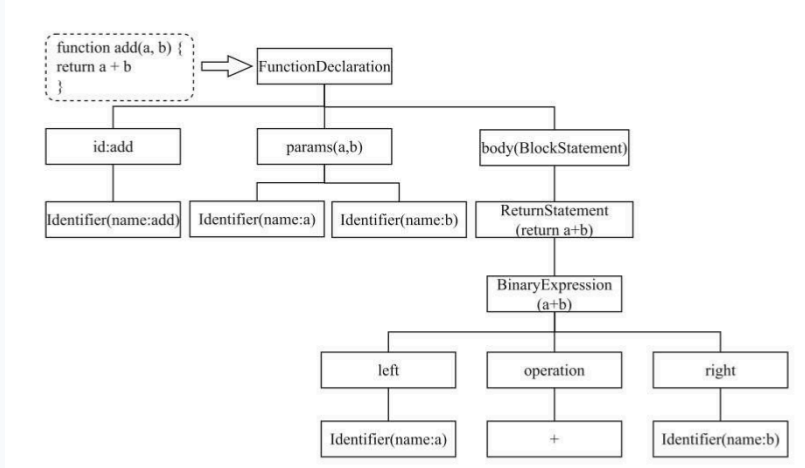
Figure 6: AST Representation

Source Code:

```
function add(a, b) {
  return a + b
}
```

AST Structure:

- FunctionDeclaration
- id: add
- params: (a, b)



Tree-Based: Key Methods

Code2Vec (Alon et al., 2019b)

- Uses AST paths to represent code
- Suitable for large-scale codebases with simple structures
- Struggles with complex structures or long-distance dependencies

Code2Seq (Alon et al., 2019a)

- Randomly samples AST paths
- Better captures diversity
- Introduces uncertainty and computational efficiency issues

Tree-Based: More Methods

SBT - Structural-Based Traversal (Hu et al., 2018a)

- Uses pairs of parentheses to represent AST structure
- Preserves complete hierarchical information
- May generate excessively long sequences

Zhang et al. (2019)

- Splits AST into subtrees
- Recursive encoding
- Better for complex local structures
- Higher computational costs

Tree-Based: AST Linearization

What is Linearization?

- Converts tree structure to a sequence
- Simplifies data processing
- Can be fed to sequence models (RNN, Transformer)

Limitations:

- Increases sequence length
- Higher memory usage
- Reduced training efficiency
- Requires language-specific design

Example - SBT Linearization:

Code: `return a + b`
AST Linearized: `( ReturnStmt ( BinaryExpr ( Identifier a ) + ( Identifier b ) ) )`

Tree-Based: Non-Linearization (Tree-based Processing)

What is Non-Linearization?

- Retains original tree structure
- Uses Tree-LSTM or GNN encoders
- Preserves parent-child relationships

Advantages:

- Reflects source code syntax directly
- Better for complex logic
- Captures structural semantics

Example - Tree-LSTM Processing:

Code: `return a + b`

```
Tree structure preserved: ReturnStmt → BinaryExpr → (Identifier: a, Op: +, Identifier: b)
Each node encoded with children's hidden states
```

Tree-Based: Advantages

Strengths:

- **Captures hierarchical structure** - better than token-based
- **Handles long-range dependencies** - through attention mechanisms
- **Preserves syntax** - maintains code structure
- **Semantic understanding** - better comprehension of code logic

Tree-Based: Limitations

Weaknesses:

- **Computational complexity** - especially for deep structures
- **Cross-function dependencies** - limited representation
- **Scalability issues** - with very large codebases
- **Memory overhead** - for linearized representations

2.3 Graph-Based Modeling

Graph-Based Modeling: Concept

Approach: Uses graph structures to capture complex semantic information in code.

Graph Types:

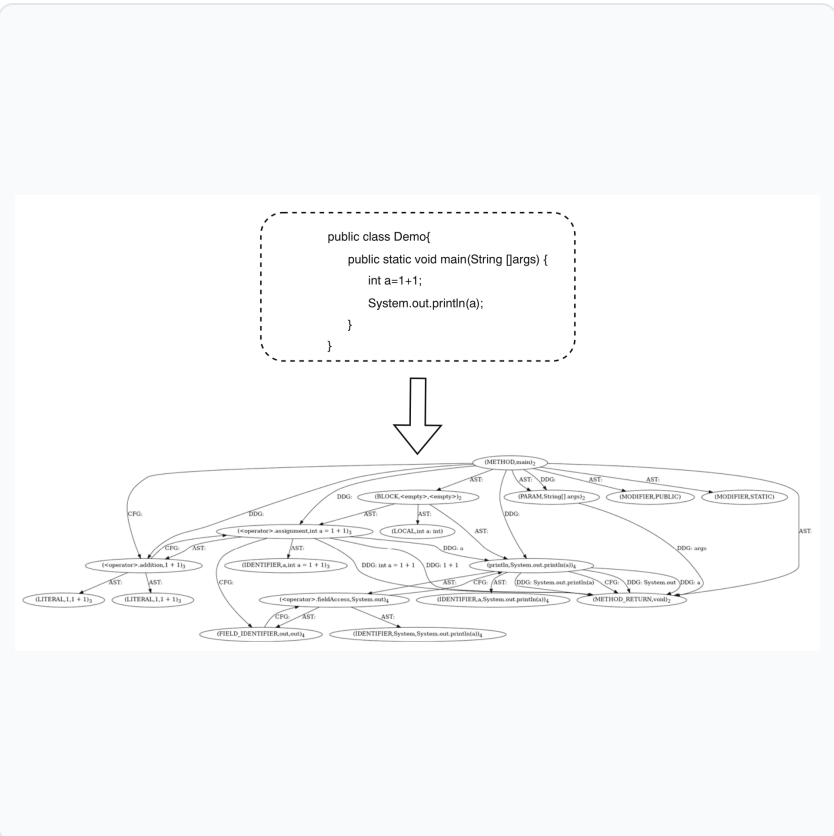
- **CFG** - Control Flow Graph
- **DFG** - Data Flow Graph
- **PDG** - Program Dependency Graph
- **CPG** - Code Property Graph

Graph-Based Example

Captures:

- Control flow relationships
- Data flow dependencies
- Program dependencies
- Multiple perspectives simultaneously

Complexity: More expressive but computationally intensive



Graph-Based: Key Methods

Graph Neural Networks (GNN)

- LeClair et al. (2020) - Graph-based neural architecture
- Liu et al. (2021a) - Combined code representations
- Wang et al. (2022b) - Graph attention network

API Context Graphs

- Yang et al. (2023b) - API usage knowledge to construct context graphs
- Graph attention mechanism for encoding

Graph-Based: Advantages

Strengths:

- **Captures complex relationships** - data flow, control flow, dependencies
- **Multi-perspective representation** - combines different graph types
- **Semantic richness** - better understanding of code semantics
- **Handles complex structures** - nested and interconnected code

Graph-Based: Limitations

Weaknesses:

- **Computational overhead** - graph construction and processing
- **Scalability challenges** - with large codebases
- **Implementation complexity** - requires graph analysis tools
- **Memory requirements** - for large graphs

2.4 Combination-Based Modeling

Combination-Based Modeling: Concept

Approach: Combines multiple modeling techniques to leverage strengths of each approach.

Common Combinations:

- Token + Tree (AST)
- Tree + Graph
- Token + Graph
- All three approaches

Combination-Based: Key Methods

Guo et al. (2021)

- Partial AST information + data flow structure
- Reduces computational overhead
- Suitable for large-scale codebases
- May lose fine-grained syntactic information

Multi-modal Approaches

- Combine token sequences, AST, and graph structures
- Use attention mechanisms to weight different representations
- Better overall code understanding

Source Code Modeling: Comparison

Approach	Strengths	Limitations
Token	Simple, scalable, language-agnostic	Loses structure, surface-level only
Tree	Preserves hierarchy, syntax-aware	Complex, memory-intensive
Graph	Rich semantics, multi-perspective	High overhead, scalability issues
Combination	Best of all worlds	Implementation complexity

Part 3: Code Summarization Generation

Generation Methods: Overview

Three Main Approaches:

- **Template-based** - Rule-based, heuristic templates
- **Information Retrieval-based** - Extract from similar code
- **Deep Learning-based** - Neural network generation

Evolution: From rule-based → IR-based → Deep learning (now dominant)

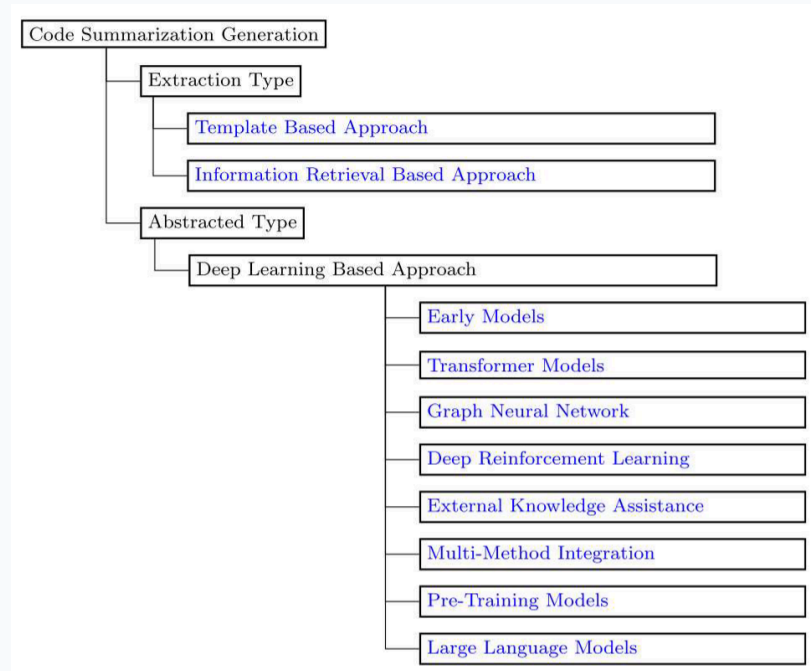
Generation Methods Overview

Template-Based:

- Early approach
- Rule-based generation
- Limited flexibility

IR-Based:

- Find similar code
- Extract summaries
- Quality depends on corpus

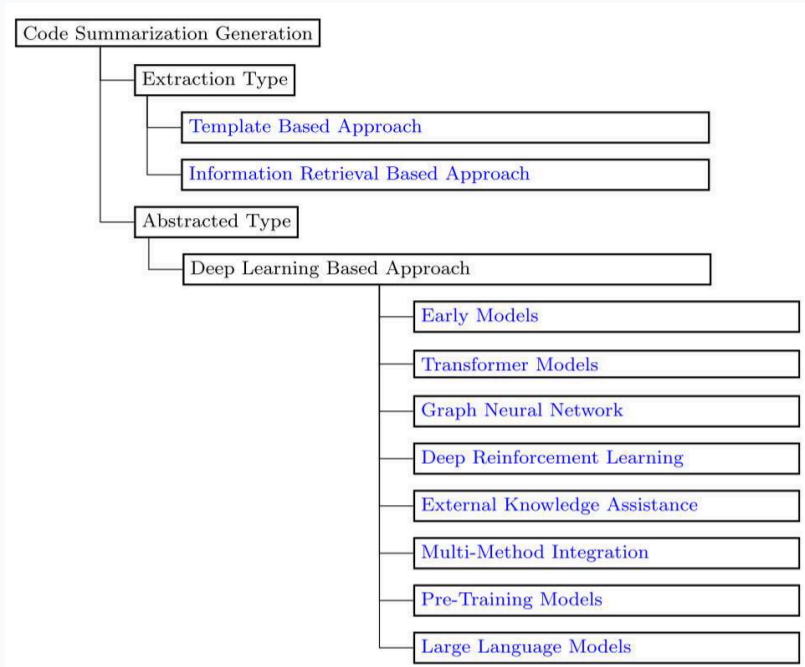


Generation Methods Overview (Continued)

Deep Learning:

- Seq2Seq, Transformer
- GNN, RL, Pre-training
- LLMs (recent)

Current Trend: Deep learning approaches now dominate, with LLMs showing promising results



3.1 Template-Based Approach

Template-Based: Overview

Approach: Uses predefined templates and heuristic rules to generate code summaries.

Process:

- Source code undergoes syntax analysis
- Key information extracted (method names, parameters, return types)
- Templates filled with extracted information
- Summary generated following template structure

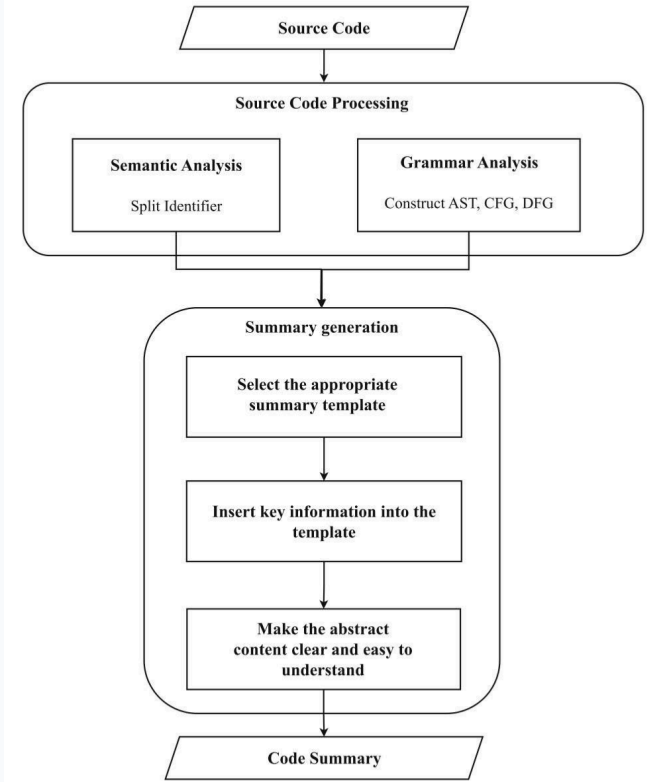
Template-Based Process

Steps:

- Syntax analysis
- Information extraction
- Template selection
- Template filling
- Summary generation

Limitations:

- Rigid structure
- Limited expressiveness
- Requires manual template design



Template-Based: Characteristics

Limitations:

- Poor handling of complex code
- Limited accuracy
- Requires extensive rule engineering
- Not adaptable to different coding styles

Use Cases:

- Simple, well-structured code
- Domain-specific applications
- When consistency is more important than flexibility

3.2 Information Retrieval-Based Approach

IR-Based: Overview

Approach: Finds similar code in a corpus and extracts/adapts their summaries.

Process:

- Construct corpus of code-summary pairs
- Given new code, find similar code in corpus
- Extract or adapt summary from similar code
- Generate final summary

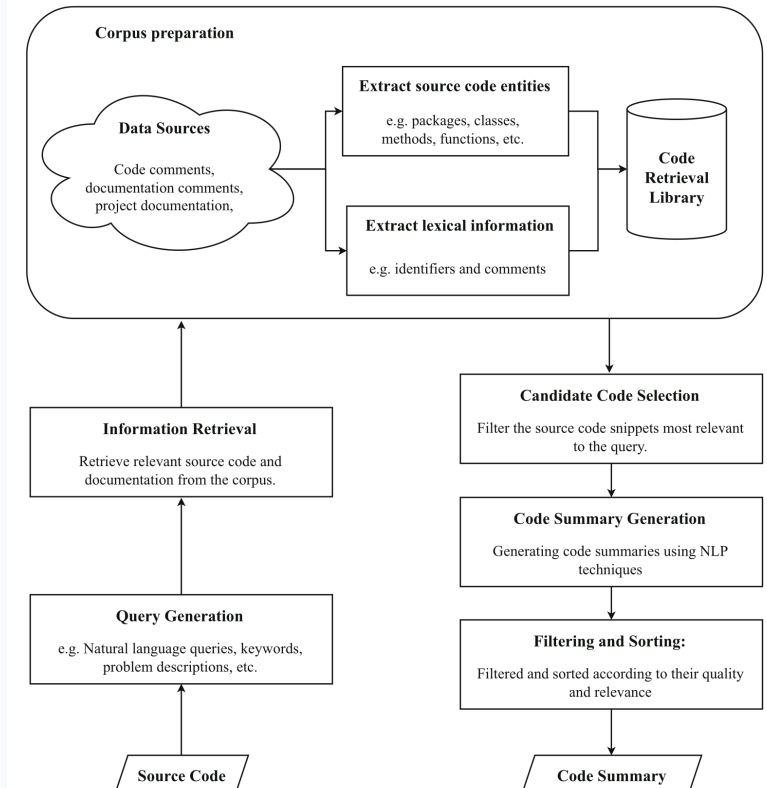
IR-Based Process

Key Components:

- Corpus construction
- Similarity matching
- Summary extraction/adaptation
- Quality depends on corpus quality

Challenges:

- Requires high-quality corpus
- Similarity matching accuracy
- Adaptation quality



IR-Based: Characteristics

Advantages:

- Can leverage existing high-quality summaries
- No training required
- Interpretable results

Limitations:

- Effectiveness depends on corpus quality
- May not find suitable matches
- Limited generalization
- Adaptation can be challenging

3.3 Deep Learning-Based Approach

Deep Learning: Overview

Approach: Uses neural network models (encoder-decoder framework) to learn mapping from source code to natural language summaries.

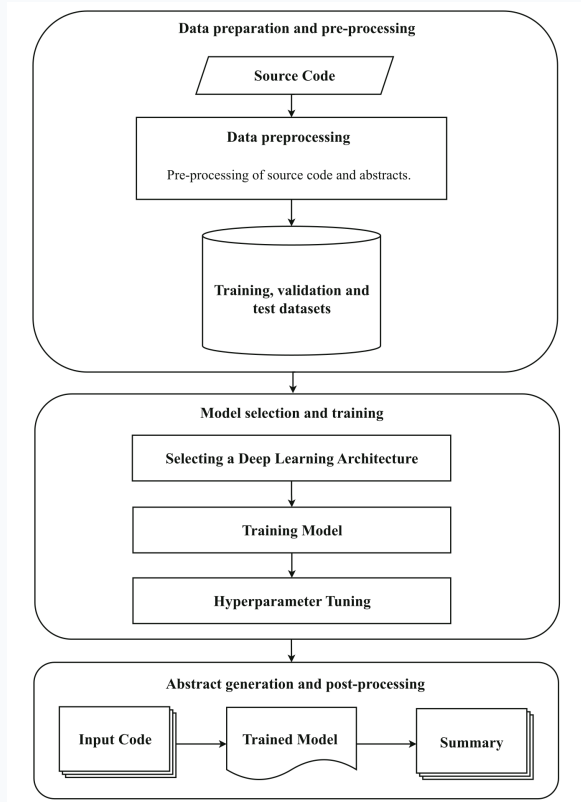
Dominant Method: Most current research focuses on deep learning approaches

Deep Learning Process

Figure 11: Basic Flowchart

Steps:

- Data preparation and preprocessing
- Model selection and training
- Hyperparameter tuning
- Evaluation
- Summary generation
- Post-processing



Deep Learning: Early Models

CODE-NN (Iyer et al., 2016)

- First neural approach
- LSTM encoder-decoder
- Attention mechanism
- Baseline for many studies

DeepCom (Hu et al., 2018a)

- AST-based analysis
- SBT (Structural-Based Traversal)
- Java method summarization

Deep Learning: Transformer Models

Transformer Architecture (Vaswani et al., 2017)

- Self-attention mechanism
- Handles long-distance dependencies
- Parallel computation
- No sequential processing needed

CodeBERT (Feng et al., 2020)

- Bimodal pre-trained model
- Code and natural language alignment
- Fine-tuning for downstream tasks

Deep Learning: Transformer Variants

Ahmad et al. (2020)

- Transformer encoder for code tokens
- Transformer decoder for summary generation
- Pairwise relationship modeling

Wu et al. (2021)

- Multi-view structure attention
- Token view, AST view, path view
- Better structure information learning

Deep Learning: Graph Neural Networks

GNN for Code Structure:

- LeClair et al. (2020) - Graph-based neural architecture
- Liu et al. (2021a) - Combined representations
- Wang et al. (2022b) - Graph attention network

Advantage:

- Better capture of code structure
- Models complex dependencies
- Leverages graph representations

Deep Learning: Reinforcement Learning

RL Approaches:

- Wan et al. (2018) - AST + sequential content in RL framework
- Wang et al. (2022a) - Hierarchical attention network with RL
- Huang et al. (2020) - Actor-critic algorithm for block comments
- Zhang et al. (2023a) - BLEU score as reward function

Goal: Improve readability and informativeness of generated summaries

Deep Learning: Pre-trained Models

CodeBERT (2020)

- Bimodal pre-training
- CodeSearchNet dataset
- Bidirectional encoders

CodeT5 (2021)

- Text encoders for code
- CodeXGLUE dataset
- Cross-language tasks

GraphCodeBERT (2021)

- Data flows as objectives
- Avoids deep AST overhead
- Graph encoder for code

Deep Learning: Large Language Models

LLM Applications:

- Ahmed & Devanbu (2022a) - Codex with few-shot learning
- Sun et al. (2023) - ChatGPT for code summarization
- Su & McMillan (2024) - GPT-3.5 knowledge distillation

Advantages:

- Powerful language understanding
- Strong transfer learning
- Few-shot/zero-shot capabilities

Deep Learning: Integration Methods

Retrieval-Augmented Generation:

- Zhang et al. (2020) - Hybrid retrieval and generation
- LeClair et al. (2021) - Ensemble of RNN, Transformer, GNN
- Sun et al. (2024) - Extraction + abstraction fusion

Goal: Combine strengths of different approaches

Deep Learning: Challenges

Current Limitations:

- **Computational resources** - Large-scale parallel computing needed
- **Out-of-vocabulary words** - Low-frequency tokens, proper nouns
- **Interpretability** - "Black box" nature of neural networks
- **Fluency** - Not yet comparable to natural language

Deep Learning: Future Directions

Research Areas:

- **Auxiliary information** - API knowledge, documentation, test cases
- **Efficient architectures** - Lighter models, faster optimization
- **Interpretability** - Attention visualization, error detection
- **LLM integration** - Better adaptation to code summarization

Part 4: Quality Evaluation

Quality Evaluation: Overview

Challenge: No unified standard for what constitutes "best" code comments

Two Main Approaches:

- **Manual Evaluation** - Human reviewers assess quality
- **Automatic Evaluation** - Metrics-based assessment

Best Practice: Combination of both approaches

4.1 Manual Evaluation

Manual Evaluation: Overview

Definition: Inviting experienced developers to assess code summaries through detailed scoring or ranking.

Characteristics:

- Highly subjective
- Time-consuming and labor-intensive
- Provides valuable depth and granularity
- Complements automatic evaluation

Manual Evaluation: Metrics

Primary Evaluation Metrics:

- **Readability** - Fluency and ease of understanding
- **Simplicity** - Concise summarization of main functions
- **Relevance** - Match between summary and code functionality
- **Completeness** - Comprehensive coverage of code functions
- **Correctness** - Accuracy without errors or misleading descriptions

Manual Evaluation: Methods

Expert Evaluation:

- Conducted by developers/researchers
- Emphasizes accuracy and professionalism
- May be subject to expert subjectivity

User Evaluation:

- Conducted by actual code users
- Reflects practical effectiveness
- May exhibit bias due to user backgrounds

Manual Evaluation: Modern Approaches

Rating Scales:

- Quantify summaries based on predefined criteria
- Reduce subjectivity
- Enhance consistency

Open-ended Feedback:

- Detailed written comments
- Reveal potential issues
- Suggest improvements

Manual Evaluation: Challenges

Common Issues:

- **Subjectivity** - Different reviewers, different criteria
- **High Cost** - Time and human resources
- **Consistency** - Ensuring uniform evaluation

Solutions:

- Multiple reviewers + statistical methods (Kappa)
- Rational sample selection
- Automatic evaluation for initial screening
- Training and coordination

4.2 Automatic Evaluation

Automatic Evaluation: Overview

Approach: Uses metrics from machine translation and text summarization to evaluate text similarity between generated and reference summaries.

Two Types:

- **Machine Translation Metrics** - BLEU, METEOR, ROUGE, CIDER
- **Statistical Metrics** - Precision, Recall (less common)

BLEU Metric

BLEU (Bilingual Evaluation Understudy)

- Measures word choice and fluency
- Based on n-gram overlap
- BLEU-1/2/3/4 for different n-gram levels

Limitations:

- Doesn't account for semantic correctness
- Influenced by common words
- Favors short sentences

Equation (1): BLEU Score

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \cdot \log p_n\right)$$

Where:

- BP = Brevity Penalty
- p_n = n-gram precision
- w_n = weight (usually $1/N$)
- N = maximum n-gram order

METEOR Metric

METEOR (Metric for Evaluation of Translation with Explicit ORdering)

- Word-based metric
- Uses recall to capture coverage
- Word matching between generated and reference

Advantage: Uses WordNet to expand synonym sets, better semantic evaluation

METEOR Equations:

$$\text{METEOR} = F_{mean} \cdot (1 - Pen)$$

$$F_{mean} = \frac{P \cdot R}{\alpha P + (1-\alpha)R} \quad P = \frac{m}{c} \quad R = \frac{m}{r}$$

$$Pen = \gamma \cdot \left(\frac{ch}{m}\right)^{\beta}$$

Variables: m = matched unigrams, c = candidate length, r = reference length, ch = chunks, α, γ, β = tunable parameters

ROUGE Metric

ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

- Focuses on recall (coverage)
- ROUGE-N, ROUGE-L, ROUGE-W, ROUGE-S variants
- Most common: ROUGE-L (LCS)

Limitation: Doesn't account for fluency

ROUGE Equations:

$$P_{lcs} = \frac{LCS(X,Y)}{m} \quad R_{lcs} = \frac{LCS(X,Y)}{n}$$

$$F_{Rouge_{lcs}} = \frac{(1+\beta^2) \cdot P_{lcs} \cdot R_{lcs}}{R_{lcs} + \beta^2 \cdot P_{lcs}}$$

Variables: LCS = longest common subsequence, m = candidate length, n = reference length, β = weight parameter

CIDER Metric

CIDER (Consensus-based Image Description Evaluation)

- Uses TF-IDF weights and cosine similarity
- Averages scores of different n-grams
- Emphasizes word importance

Advantage: Also suitable for code summarization evaluation

CIDER Equation (11):

$$\text{CIDEr}_n(c_i, S_i) = \frac{1}{m} \sum_j \frac{g^n(c_i) \cdot g^n(s_{ij})}{\|g^n(c_i)\| \|g^n(s_{ij})\|}$$

$$\text{CIDEr}(c_i, S_i) = \sum_{n=1}^{N} w_n \cdot \text{CIDEr}_n(c_i, S_i)$$

Variables: g^n = TF-IDF weighted n-gram vector, c_i = candidate, S_i = reference set, w_n = weights (uniform $1/N$)

Automatic Evaluation: Limitations

Major Issues:

- **N-gram matching focus** - Overlooks semantics and fluency
- **Inconsistent BLEU usage** - BLEU-1 vs BLEU-4 vs BLEU-N
- **Poor correlation** - Doesn't reflect human judgment well
- **Reference dependency** - Quality depends on reference summaries

Automatic Evaluation: Improvements

New Metrics:

- **CodeBLEU** (Ren et al., 2020) - Combines code syntax and semantics
- **BLEU-DC** (Shi et al., 2022b) - Best BLEU variant for human perception
- **BERTScore** - Contextual embeddings, high correlation with humans

Future: Need metrics that assess connection between summaries and code, not just summary-reference similarity

Practical Example: Computing Metrics

Code:

```
public int add(int a, int b) { return a + b; }
```

Reference Summary:

"Returns the sum of two integers a and b"

Generated Summary:

"Adds two numbers and returns the result"

Practical Example: Metric Results

Results:

Metric	Score	Explanation
BLEU-4	0.12	Low n-gram overlap ("returns", "two")
METEOR	0.45	Captures synonyms (sum ↔ adds)
ROUGE-L	0.38	LCS: "two" ... "returns"
CIDEr	0.52	TF-IDF weights programming terms

Libraries to Compute Metrics:

- **NLTK**: `nltk.translate.bleu_score`
- **METEOR**: `nltk.translate.meteor_score`
- **ROUGE**: `rouge-score` (Google)
- **CIDEr**: `pycocoevalcap`
- **All-in-one**: `evaluate` (HuggingFace)

Evaluation: Best Practices

Combined Approach:

- Use automatic evaluation for rapid screening
- Follow with manual evaluation for detailed assessment
- Leverage efficiency of automatic + depth of manual

Current State: Neither approach is ideal - need more effective evaluation rubrics

Discussion & Future Directions

Current Challenges

Key Issues:

- **Semantic understanding** - Especially with domain knowledge and context
- **Multi-granularity problem** - Choosing appropriate information level
- **Data bias** - Generated summaries too inclined to training patterns
- **Evaluation standards** - Lack of unified criteria

Future Research Directions (1)

1. Enhancing Source Code Information Utilization

- Better use of comments, documentation, test cases, API calls
- Extract useful information from cluttered comments
- Leverage documentation and API calls for readability

2. Multi-Granularity Information

- Function-level vs module-level vs project-level
- Choose appropriate granularity for different tasks
- Merge and transform between granularities

Future Research Directions (2)

3. Integrating Deep and Non-Deep Models

- Combine template/IR with deep learning
- Balance collaboration and competition
- Leverage strengths of each approach

4. Large Language Models (LLMs)

- Explore LLM potential for code summarization
- Few-shot/zero-shot learning
- Real-time generation in development environments

Future Research Directions (3)

5. Unified Corpus and Evaluation Criteria

- Build larger-scale, higher-quality datasets
- Standardize evaluation metrics
- Combine manual and automatic evaluation advantages
- Develop domain/language-specific criteria

Key Takeaways

ASCS Pipeline:

- **Data Collection** - Foundation for quality
- **Source Code Modeling** - Token, Tree, Graph, Combination
- **Generation** - Template, IR, Deep Learning (dominant)
- **Evaluation** - Manual + Automatic (combined best)

Key Takeaways (Continued)

Current State:

- Deep learning dominates research
- LLMs showing promise
- Quality evaluation remains challenging
- Need for better datasets and metrics

Open Questions:

- How to better utilize code information?
- How to handle multi-granularity?
- How to improve evaluation?
- How to integrate different approaches?

Thank You!

Questions?

References:

- Zhang et al. (2024). "A review of automatic source code summarization." Empirical Software Engineering, 29:162

Contact: fabio.pinelli@imtlucca.it