A New Approach to Evaluating Nullability Inference Tools

Nima Karimipour[*1], Erfan Arvan[*2],
Martin Kellogg[2], Manu Sridharan[1]
* Equal contribution
[1]University of California, Riverside
[2]New Jersey Institute of Technology

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
    - “Testing shows the presence of bugs, not their absence”

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
    - “Testing shows the presence of bugs, not their absence”
- To scale to real programs, many verifiers are **modular**
    - Downside: humans must **write specifications**
        - Hard for legacy code

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
  - “Testing shows the presence of bugs, not their absence”
- To scale to real programs, many verifiers are **modular**
  - Downside: humans must **write specifications**
    - Hard for legacy code
- **Pluggable typecheckers** extend a host type system

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
    - "Testing shows the presence of bugs, not their absence"
- To scale to real programs, many verifiers are **modular**
    - Downside: humans must **write specifications**
        - Hard for legacy code
- **Pluggable typecheckers** extend a host type system
    - Prior work has introduced **type inference techniques** to try to solve this problem

Our Goal Today: A Fair Comparison

- Three high-level research questions:

Our Goal Today: A Fair Comparison

- Three high-level research questions:
    - What’s the **right methodology** to compare inference tools?

Our Goal Today: A Fair Comparison

- Three high-level research questions:
  - What's the **right methodology** to compare inference tools?
  - Which extant tool is the **state-of-the-art**?

Our Goal Today: A Fair Comparison

- Three high-level research questions:
  - What's the **right methodology** to compare inference tools?
  - Which extant tool is the **state-of-the-art**?
  - What **opportunities for improving** the state-of-the-art exist?

Our Goal Today: A Fair Comparison

- Three high-level research questions:
  - What's the **right methodology** to compare inference tools?
  - Which extant tool is the **state-of-the-art**?
  - What **opportunities for improving** the state-of-the-art exist?
- Talk outline:
  - Background on the problem and prior work

Our Goal Today: A Fair Comparison

- Three high-level research questions:
    - What's the **right methodology** to compare inference tools?
    - Which extant tool is the **state-of-the-art**?
    - What **opportunities for improving** the state-of-the-art exist?
- Talk outline:
    - Background on the problem and prior work
    - Methodology issue: **bias** in type reconstruction experiments
        - Caused by how humans annotate code

Our Goal Today: A Fair Comparison

- Three high-level research questions:
    - What's the **right methodology** to compare inference tools?
    - Which extant tool is the **state-of-the-art**?
    - What **opportunities for improving** the state-of-the-art exist?
- Talk outline:
    - Background on the problem and prior work
    - Methodology issue: **bias** in type reconstruction experiments
        - Caused by how humans annotate code
    - A **fair comparison**, using an improved methodology

Our Goal Today: A Fair Comparison

- Three high-level research questions:
  - What’s the **right methodology** to compare inference tools?
  - Which extant tool is the **state-of-the-art**?
  - What **opportunities for improving** the state-of-the-art exist?
- Talk outline:
  - Background on the problem and prior work
  - Methodology issue: **bias** in type reconstruction experiments
    - Caused by how humans annotate code
  - A **fair comparison**, using an improved methodology
  - Discussion of how the state-of-the-art can improve

Background: Pluggable Types

```
int x
```

Background: Pluggable Types

```
@Positive int x
```

Background: Pluggable Types

```
@Even int x
```

Background: Pluggable Types

```
@Nullable Object x
```

Background: Pluggable Types

- **Widely adopted**
    - Uber, Meta, AWS, Google, Oracle, etc.

Background: Pluggable Types

- **Widely adopted**
  - Uber, Meta, AWS, Google, Oracle, etc.
- **Attractive to developers**
  - Familiar, high precision, sound, fast checking, modular, …

Background: Pluggable Types

- **Widely adopted**
  - Uber, Meta, AWS, Google, Oracle, etc.
- **Attractive to developers**
  - Familiar, high precision, sound, fast checking, modular, …
- Downside: **manual annotation** of legacy codebases

Background: Pluggable Types

- **Widely adopted**
  - Uber, Meta, AWS, Google, Oracle, etc.
- **Attractive to developers**
  - Familiar, high precision, sound, fast checking, modular, …
- Downside: **manual annotation** of legacy codebases
  - This is the **problem** that we’re targeting today

Background: Pluggable Types

- **Widely adopted**
  - Uber, Meta, AWS, Google, Oracle, etc.
- **Attractive to developers**
  - Familiar, high precision, sound, fast checking, modular, …
- Downside: **manual annotation** of legacy codebases
  - This is the **problem** that we're targeting today
- Recent work has proposed **3 new type inference techniques**:
  - Checker Framework Whole-Program Inference (ASE 2023)
  - NullAway Annotator (FSE 2023)
  - NullGTN (arxiv 2024)

Background: Pluggable Types

- Widely adopted
  - Uber, Meta, AWS, Google, Oracle, etc.
- **Attractive to developers**
  - Familiar, high precis …
- Downside: **manual ann**
  - This is the **problem**
- Recent work has proposed **3 new type inference techniques**:
  - Checker Framework Whole-Program Inference (ASE 2023)
  - NullAway Annotator (FSE 2023)
  - NullGTN (arxiv 2024)

Next: a brief introduction to these three extant tools

Background: CF WPI (ASE 2023)

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
    - Reduces user effort: no annotations on **local variables**

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
    - Reduces user effort: no annotations on **local variables**
    - Implemented as intra-procedural **dataflow analysis**
    - Typically implemented at the **framework level**

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement local type inference within method bodies
  - Reduces user effort: no annotations on local variables
  - Implemented as intra-procedural dataflow analysis
  - Typically implemented at the framework level
- **Key idea**: iteratively run local inference and propagate results

Background: CF WPI (ASE 2023)

- Most pluggable typecheckers already implement **local type inference** within method bodies
  - Reduces user effort: no annotations on **local variables**
  - Implemented as intra-procedural **dataflow analysis**
  - Typically implemented at the **framework level**
- **Key idea**: iteratively run local inference and propagate results
  - Advantage: works with any typechecker built on a framework "for free" (no per-typechecker code required)

Background: NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations

Background: NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
  - With some optimizations to reduce the search space

Background: NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
    - With some optimizations to reduce the search space
- Only implementation is for NullAway (FSE '19, developed at Uber)

Background: NullAway Annotator (FSE 2023)

- **Key idea**: use warnings from the checker as a **fitness function** for annotations
- Iterative, bounded-depth search for annotation set that **minimizes checker warnings**
    - With some optimizations to reduce the search space
- Only implementation is for NullAway (FSE '19, developed at Uber)
    - Only infers `@Nullable` annotations

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. *Practical Inference of Nullability Types.* In ESEC/FSE 2023.

Background: NullGTN (arxiv 2024)

- Graph-based **deep learning** model

Background: NullGTN (arxiv 2024)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations

Background: NullGTN (arxiv 2024)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would

Background: NullGTN (arxiv 2024)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would
- Trained on ~32k classes with at least one `@Nullable` annotation from GitHub
  - Data from **many sources**: checkers, documentation, etc.

Background: NullGTN (arxiv 2024)

- Graph-based **deep learning** model
  - Inspired by recent success of similar ML-based techniques for inferring Python and TypeScript type annotations
- **Key idea**: place annotations like a human would
- Trained on ~32k classes with at least one `@Nullable` annotation from GitHub
  - Data from **many sources**: checkers, documentation, etc.
- Only infers `@Nullable`

Key Differences

Key Differences

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
    - Others claim they should generalize, but it’s not evaluated

Key Difference

Implication: comparison has to be focused on nullability type systems, for now

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it’s not evaluated

Key Differences

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it’s not evaluated
- Only NullGTN can possibly annotate **entrypoint parameters**
  - (Assuming no test cases)
  - In WPI’s evaluation, this was the largest cause of missed human-written annotations (11%)

Key Differences

- CF WPI ***implementation*** is the only one that supports multiple different pluggable type systems
  - Others claim they should generalize, but it's not evaluated
- Only NullGTN can possibly annotate **entrypoint parameters**
  - (Assuming no test cases)
  - In WPI's evaluation, this was the largest cause of missed human-written annotations (11%)
- All three tools were evaluated separately
  - 2/3 (WPI, NullGTN) use "**type reconstruction**" experiments
    - NullAway Annotator evaluation lacks ground truth

Type Reconstruction Experiments

Methodology:

Type Reconstruction Experiments

Methodology:

- Collect benchmarks **previously annotated** by humans

Type Reconstruction Experiments

Methodology:

- Collect benchmarks **previously annotated** by humans
- **Remove** annotations

Type Reconstruction Experiments

Methodology:

- Collect benchmarks **previously annotated** by humans
- **Remove** annotations
- Run inference

Type Reconstruction Experiments

Methodology:

- Collect benchmarks **previously annotated** by humans
- **Remove** annotations
- Run inference
- **Compare** inference results to human-written annotations

Type Reconstruction Experiments

Methodology:

- Collect benchmarks **previously annotated** by humans
- **Remove** annotations
- Run inference
- **Compare** inference results to human-written annotations

Major advantage: have ground truth: the human-written annotations

Type Reconstruction Experiments: Biased!

- Recall our motivation: we want to use inference to annotate **never-annotated** programs

Type Reconstruction Experiments: Biased!

- Recall our motivation: we want to use inference to annotate **never-annotated** programs
    - But type reconstruction benchmarks aren't "never-annotated"
        - In fact, they differ in important ways!

Type Reconstruction Experiments: Biased!

- Recall our motivation: we want to use inference to annotate **never-annotated** programs
  - But type reconstruction benchmarks aren't "never-annotated"
    - In fact, they differ in important ways!
- **Intuition**: programmers **change semantics** as they annotate
  - E.g., add null checks, work around false positives

Type Reconstruction Experiments: Biased!

- Recall our motivation: we want to use inference to annotate **never-annotated** programs
  - But type reconstruction benchmarks aren't "never-annotated"
    - In fact, they differ in important ways!
- **Intuition**: programmers **change semantics** as they annotate
  - E.g., add null checks, work around false positives
- These changes could **simplify inference**
  - We can check this empirically

Type Reconstruction Experiments: Biased!

Methodology:

Type Reconstruction Experiments: Biased!

Methodology:

- Collect **before** and **after** versions of human-annotated benchmarks
    - Via per-project historical investigation of git history

Type Reconstruction Experiments: Biased!

Methodology:

- Collect **before** and **after** versions of human-annotated benchmarks
    - Via per-project historical investigation of git history
- **Manually categorize** changes

Type Reconstruction Experiments: Biased!

Methodology:

- Collect **before** and **after** versions of human-annotated benchmarks
    - Via per-project historical investigation of git history
- **Manually categorize** changes
- **Run inference** on both version and compare results

Type Reconstruction Experiments: Biased!

Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC

Type Reconstruction Experiments: Biased!

Results:

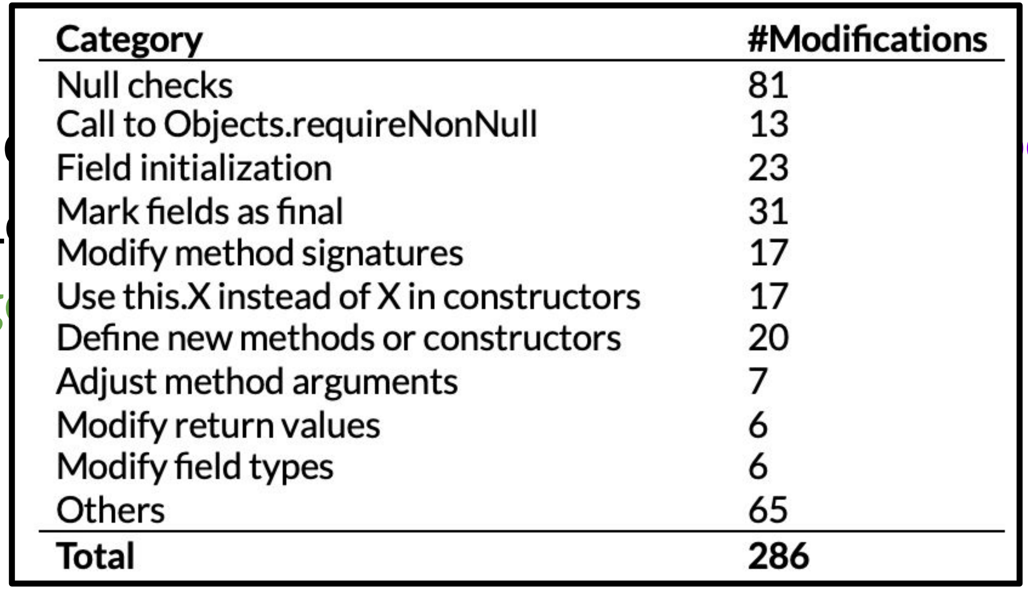
- We could identify before and after versions of **10 benchmarks**
    - ~36k LoC
- **286 changes** during annotation

Type Reconstruction Experiments: Biased!

Results:

- We could i
  - ~36k L
- 286 chang

enchmarks



Category	#Modifications
Null checks	81
Call to Objects.requireNonNull	13
Field initialization	23
Mark fields as final	31
Modify method signatures	17
Use this.X instead of X in constructors	17
Define new methods or constructors	20
Adjust method arguments	7
Modify return values	6
Modify field types	6
Others	65
Total	**286**

Type Reconstruction Experiments: Biased!

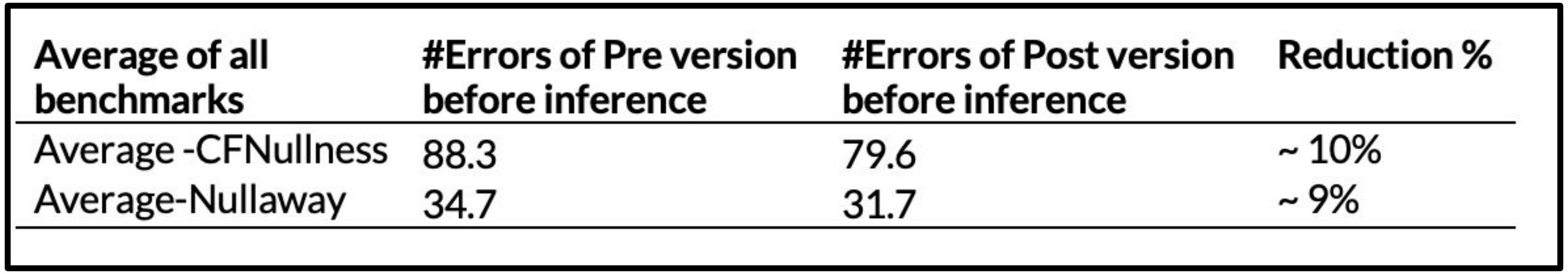
Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC
- **286 changes** during annotation
- These changes made **checking easier** (fewer warnings)

Type Reconstruction Experiments: Biased!

Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC
- **286 changes** during annotation
- These changes made **checking easier** (fewer warnings)...



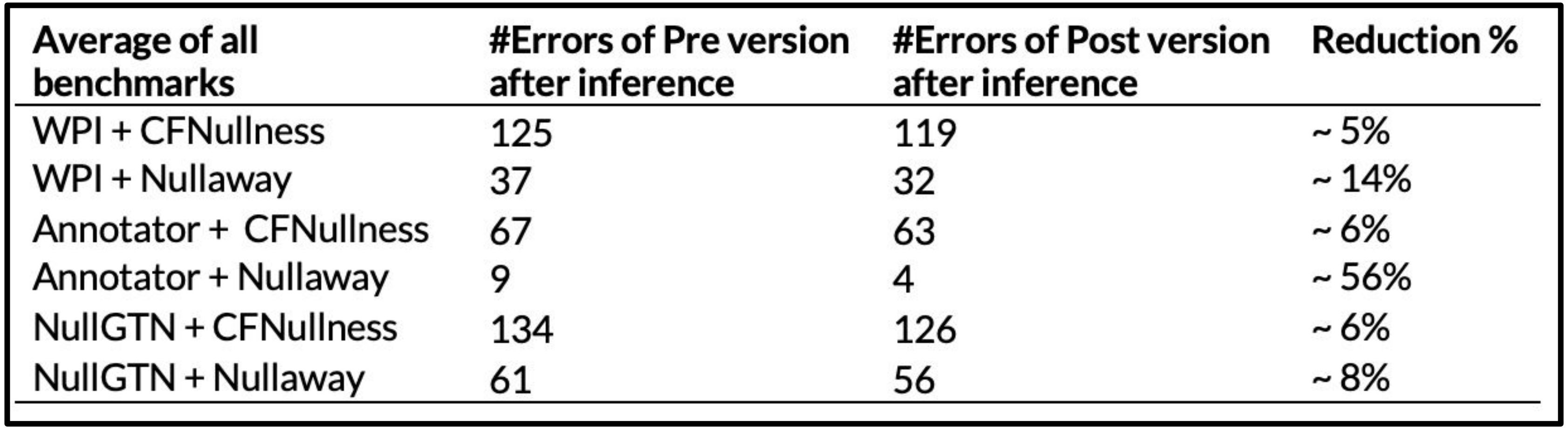
Average of all benchmarks	#Errors of Pre version before inference	#Errors of Post version before inference	Reduction %
Average -CFNullness	88.3	79.6	~ 10%
Average-Nullaway	34.7	31.7	~ 9%

Type Reconstruction Experiments: Biased!

Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC
- **286 changes** during annotation
- These changes made **checking easier** (fewer warnings)...
- ...and made **inference easier** (more warning reduction)

Type Reconstruction Experiments: Biased!



Average of all benchmarks	#Errors of Pre version after inference	#Errors of Post version after inference	Reduction %
WPI + CFNullness	125	119	~ 5%
WPI + Nullaway	37	32	~ 14%
Annotator + CFNullness	67	63	~ 6%
Annotator + Nullaway	9	4	~ 56%
NullGTN + CFNullness	134	126	~ 6%
NullGTN + Nullaway	61	56	~ 8%

- …and made **inference easier** (more warning reduction)

Type Reconstruction Experiments: Biased!

Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC
- **286 changes** during annotation
- These changes made **checking easier** (fewer warnings)...
- …and made **inference easier** (more warning reduction)

Conclusion: developers make changes beyond just writing annotations when “annotating”

Type Reconstruction Experiments: Biased!

Results:

- We could identify before and after versions of **10 benchmarks**
  - ~36k LoC
- **286 changes** during annotation
- These changes made **checking easier** (fewer warnings)...
- ...and made **inference easier** (more warning reduction)

Conclusion: developers make changes beyond just writing annotations when "annotating"

- Cannot fairly evaluate inference tools on pre-annotated code

Alternative Experimental Design

- NullAway Annotator evaluation used **warning reduction**

Alternative Experimental Design

- NullAway Annotator evaluation used **warning reduction**
  - **Problem**: warning reduction doesn't tell the whole story
    - For example, correct annotations could **add new warnings** by revealing real bugs!

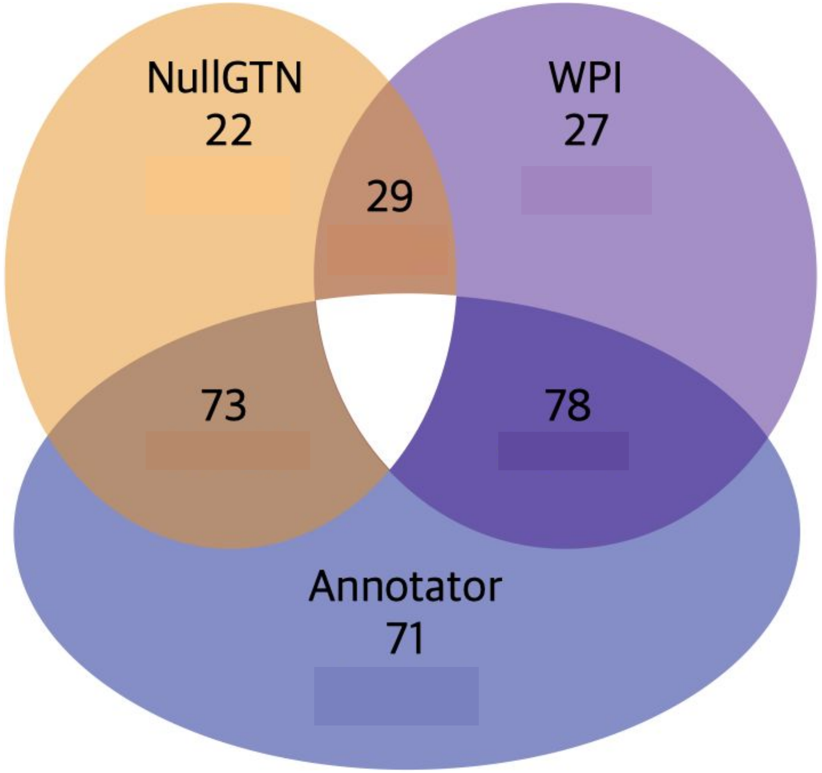
Alternative Experimental Design

- NullAway Annotator evaluation used **warning reduction**
    - **Problem**: warning reduction doesn't tell the whole story
        - For example, correct annotations could **add new warnings** by revealing real bugs!
- To fairly compare all three tools, we combined warning reduction with **manual inspection** of different annotation choices
    - Same set of **never-annotated** standard benchmarks

Alternative Experimental Design

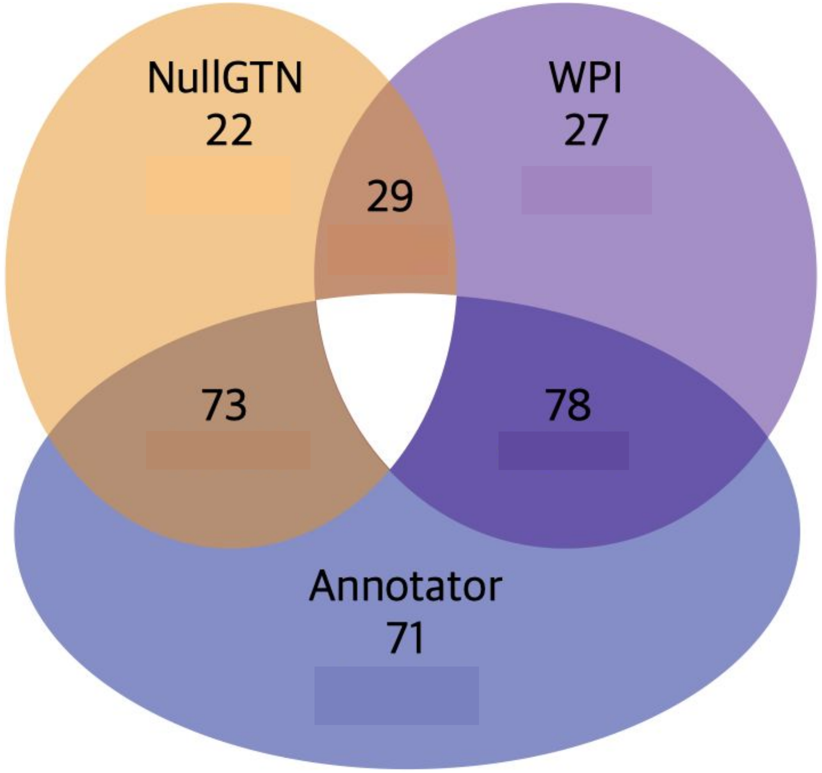
- NullAway Annotator evaluation used **warning reduction**
  - **Problem**: warning reduction doesn't tell the whole story
    - For example, correct annotations could **add new warnings** by revealing real bugs!
- To fairly compare all three tools, we combined warning reduction with **manual inspection** of different annotation choices
  - Same set of **never-annotated** standard benchmarks
  - Definition for manual evaluation: a declaration should be marked as `@Nullable` if there exists a read of it that may observe a null value

Direct Comparison: Manual Analysis



Direct Comparison: Manual Analysis

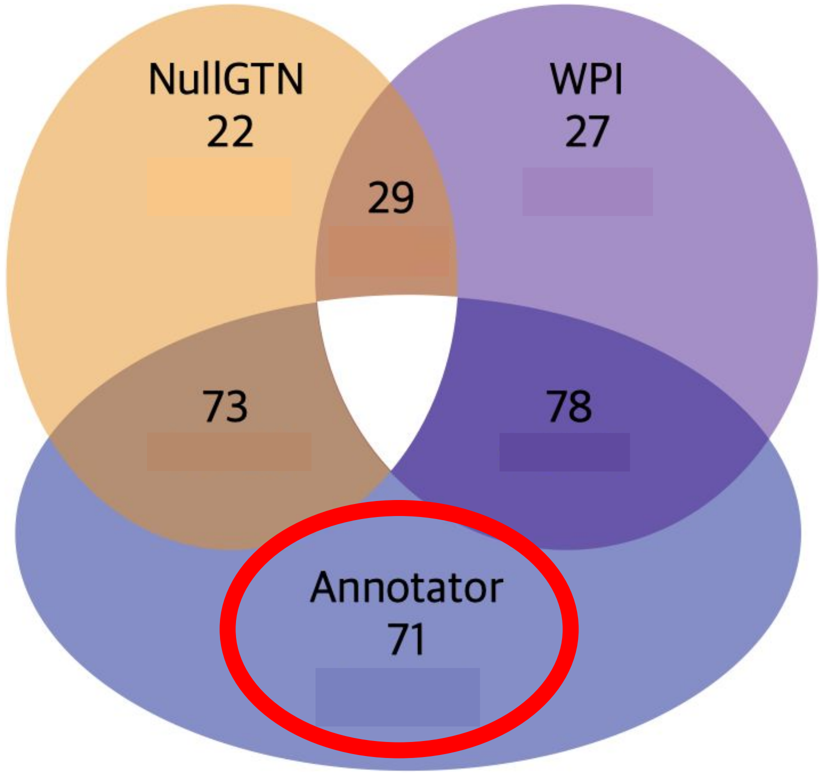
- Each number represents the number of times that the tool handles a disagreement correctly



Direct Comparison: Manual Analysis

- Each number represents the number of times that the tool handles a disagreement correctly

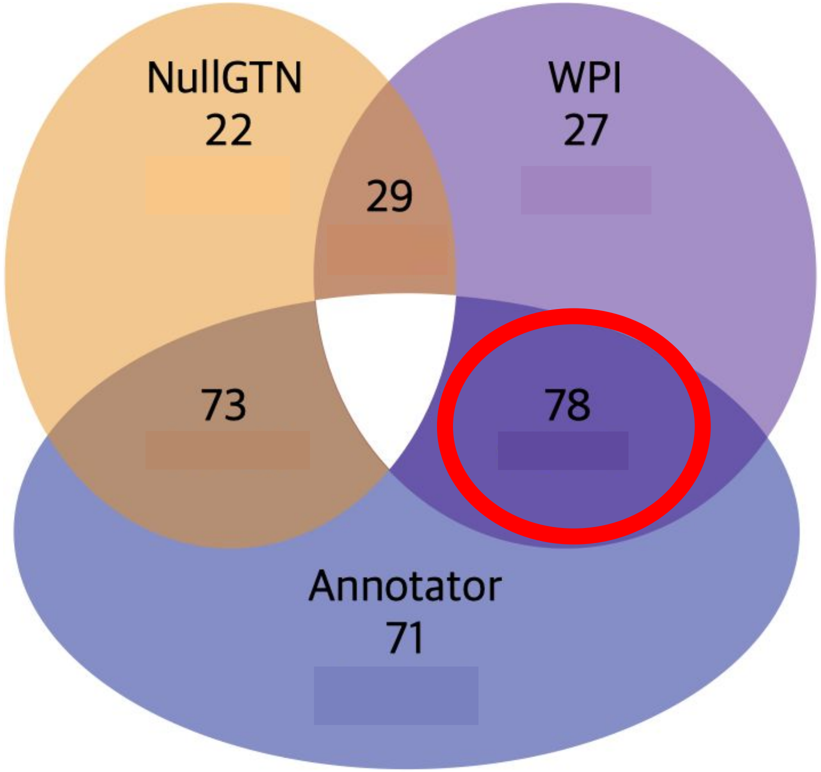
E.g., for 71 disagreements only NullAway Annotator is correct



Direct Comparison: Manual Analysis

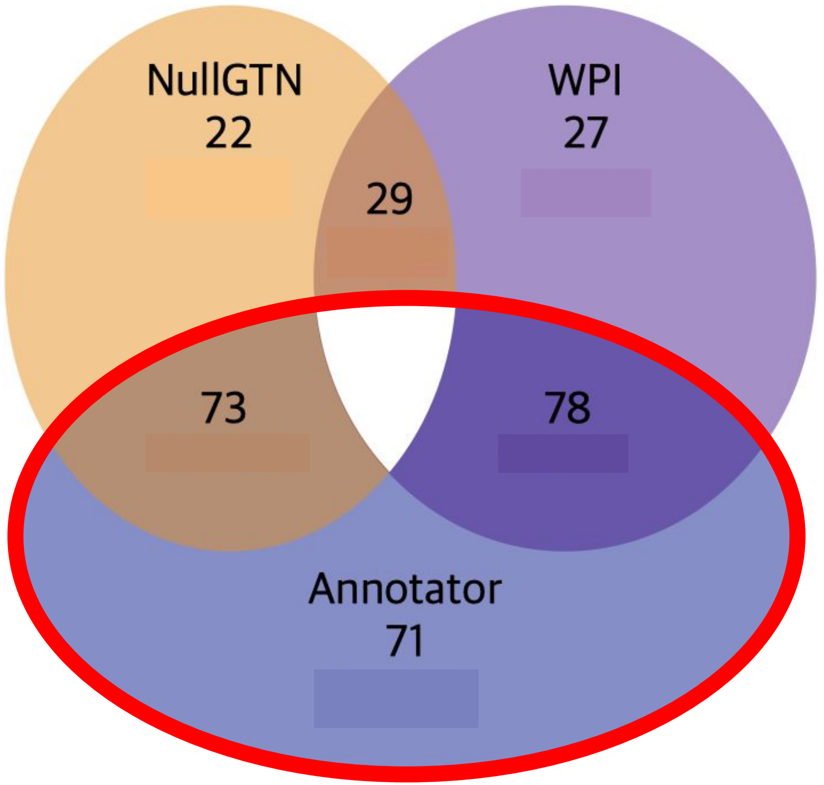
- Each number represents the number of times that the tool handles a disagreement correctly

And for 78 disagreements, both WPI and Annotator are correct (and NullGTN is wrong)



Direct Comparison: Manual Analysis

- Each number represents the number of times that the tool handles a disagreement correctly
- Overall conclusion: Annotator makes the **fewest mistakes**, but it doesn't strictly dominate the other tools



Direct Comparison: Discussion

Direct Comparison: Discussion

- Annotator makes the fewest mistakes and has the highest error reduction, but is still **far from perfect**
  - Both other tools sometimes are the only correct tool

Direct Comparison: Discussion

- Annotator makes the fewest mistakes and has the highest error reduction, but is still **far from perfect**
    - Both other tools sometimes are the only correct tool
- WPI is hampered by internal consistency
    - Inherits **dataflow imprecision** from the typechecker
        - Causes errors to **cascade**

Direct Comparison: Discussion

- Annotator makes the fewest mistakes and has the highest error reduction, but is still **far from perfect**
  - Both other tools sometimes are the only correct tool
- WPI is hampered by internal consistency
  - Inherits **dataflow imprecision** from the typechecker
    - Causes errors to **cascade**
- NullGTN overgeneralizes
  - We also observed that it handles “poorly-written” code especially badly

Future Work

Future Work

- Since developers make changes while annotating, why don't inference tools?
    - E.g., integrate **refactoring** or **automated program repair** (APR) tools with inference?

Future Work

- Since developers make changes while annotating, why don't inference tools?
    - E.g., integrate **refactoring** or **automated program repair** (APR) tools with inference?
- Even for "simple" pluggable type systems like nullability, state-of-the-art is disappointing
    - Lots of **room for improvement**

Future Work

- Since developers make changes while annotating, why don't inference tools?
  - E.g., integrate **refactoring** or **automated program repair** (APR) tools with inference?
- Even for "simple" pluggable type systems like nullability, state-of-the-art is disappointing
  - Lots of **room for improvement**
- Can we combine the strengths of different tools? E.g.:
  - Use NullGTN **only** for entrypoint parameters?
  - Could **warning fitness** stop imprecision cascades in WPI?

Summary: Pluggable Type Inference

Summary: Pluggable Type Inference

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**

Summary: Pluggable Type Inference

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**
- Previous evaluations overstated effectiveness of inference, because of **biased type reconstruction experiments**
  - Developers change their code while annotating!

Summary: Pluggable Type Inference

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**
- Previous evaluations overstated effectiveness of inference, because of **biased type reconstruction experiments**
    - Developers change their code while annotating!
- Of the extant nullability inference tools, **NullAway Annotator** produces marginally better results than the others

Summary: Pluggable Type Inference

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**
- Previous evaluations overstated effectiveness of inference, because of **biased type reconstruction experiments**
    - Developers change their code while annotating!
- Of the extant nullability inference tools, **NullAway Annotator** produces marginally better results than the others
    - But **no tool strictly dominates**, and all tools sometimes do better, so there's lots of room for improvement

Summary: Pluggable Type Inference

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**
- Previous evaluations overstated effectiveness of inference, because of **biased type reconstruction experiments**
  - Developers change their code while annotating!
- Of the extant nullability inference tools, **NullAway Annotator** produces marginally better results than the others
  - But **no tool strictly dominates**, and all tools sometimes do better, so there's lots of room for improvement
  - Future work in inference should include **refactoring/APR**

Summary: Plugg

Thanks to my co-authors: Erfan Arvan, Nima Karimipour, and Manu Sridharan

- Inference is a promising way to help developers adopt pluggable type systems, by **automating the annotation burden**
- Previous evaluations overstated effectiveness of inference, because of **biased type reconstruction experiments**
  - Developers change their code while annotating!
- Of the extant nullability inference tools, **NullAway Annotator** produces marginally better results than the others
  - But **no tool strictly dominates**, and all tools sometimes do better, so there's lots of room for improvement
  - Future work in inference should include **refactoring/APR**

Direct Comparison

- Benchmark: NJR-1 dataset [1]
    - 255 Java programs, ~1.4 million LoC

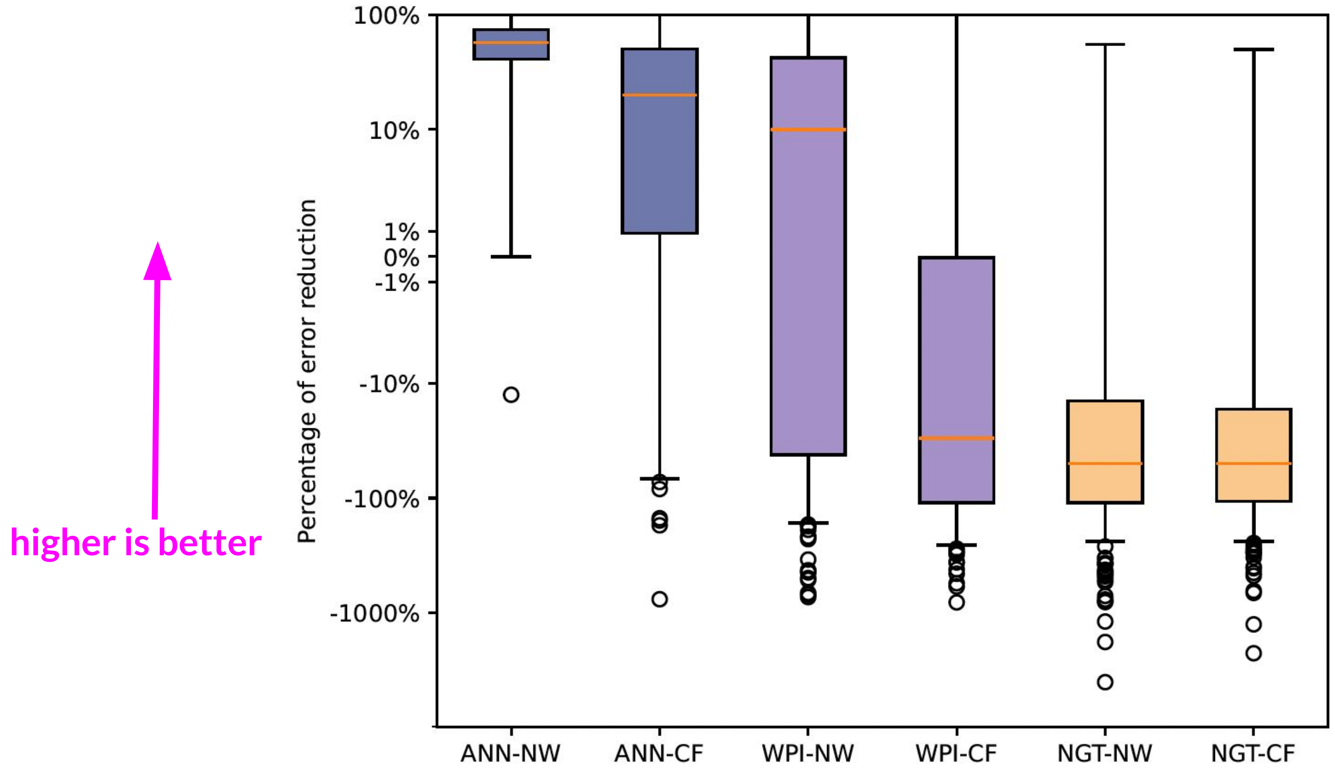
[1] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. NJR-1 dataset. https://zenodo.org/records/8015477.

Direct Comparison

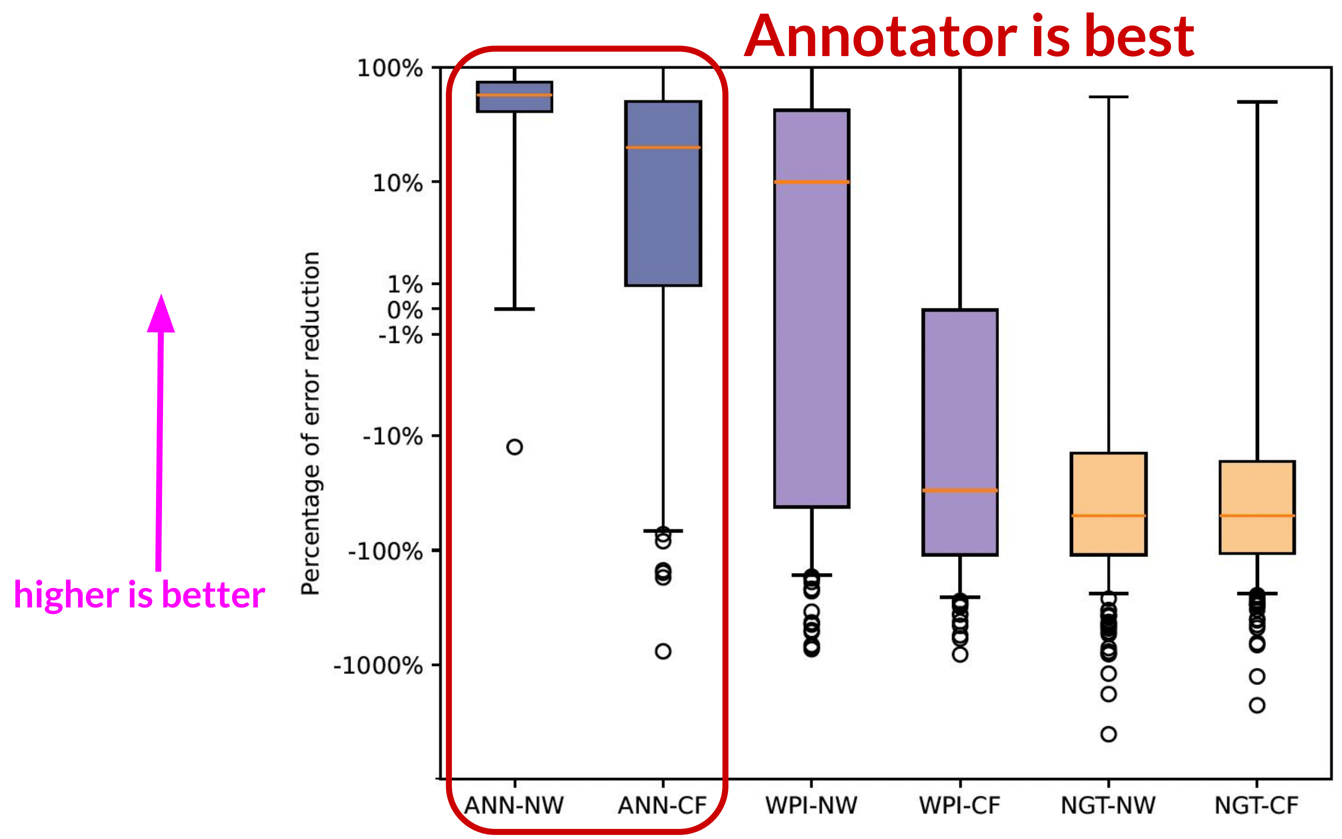
- Benchmark: NJR-1 dataset [1]
  - 255 Java programs, ~1.4 million LoC
- Two proxies for quality:
  - Manual analysis of 300 sampled disagreements
  - Warning reduction

[1] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. NJR-1 dataset. https://zenodo.org/records/8015477.

Direct Comparison: Warning Reduction



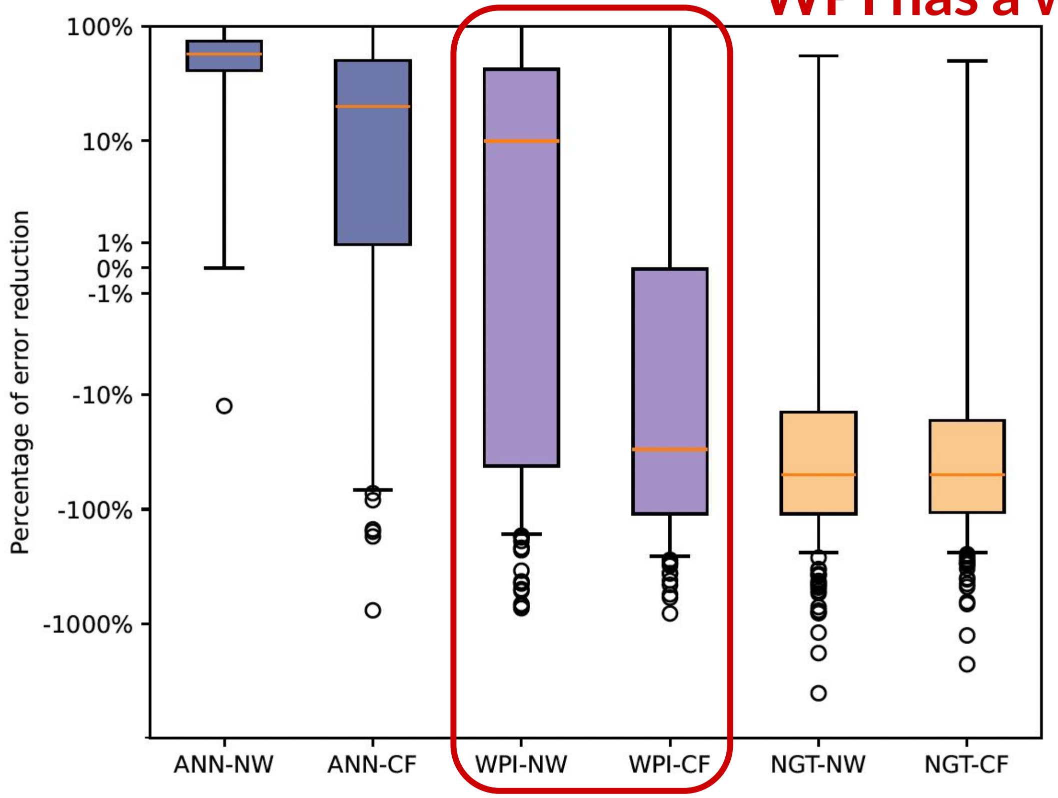
Direct Comparison: Warning Reduction



Direct Comparison: Warning Reduction

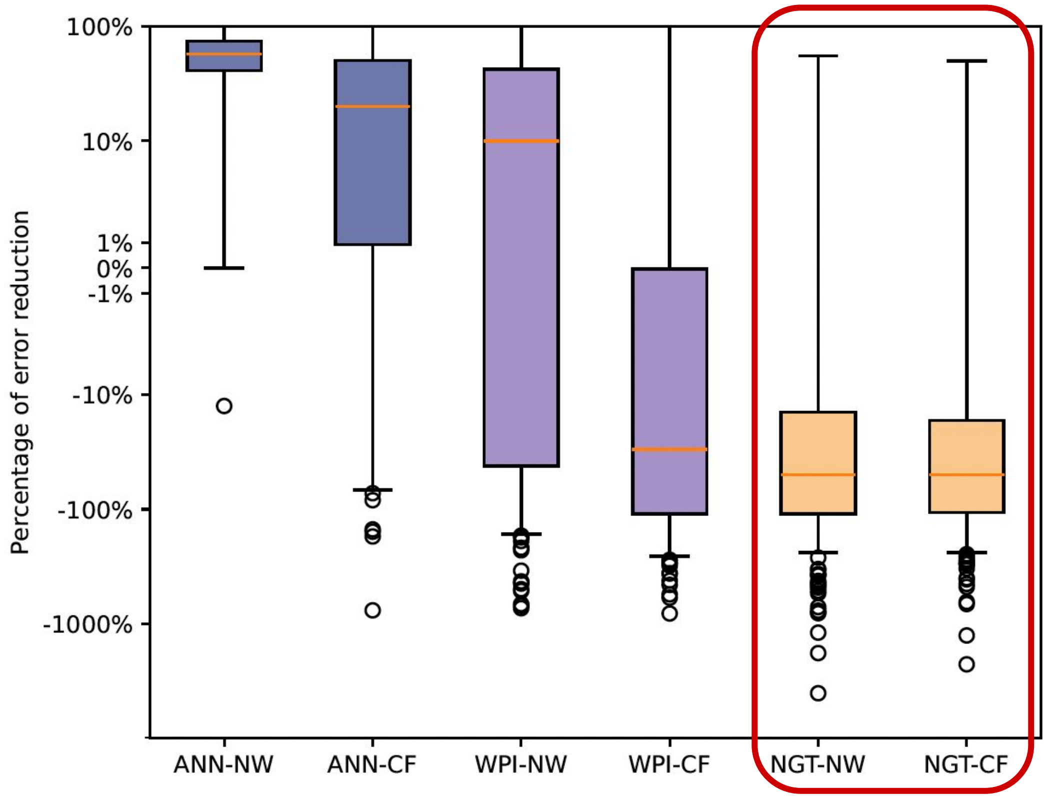
WPI has a wide range

higher is better



Direct Comparison: Warning Reduction

higher is better



NullGTN is consistently worst