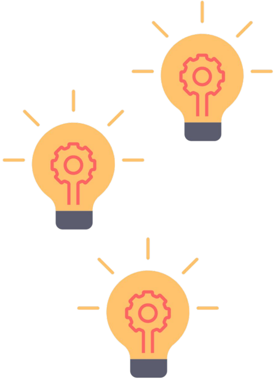
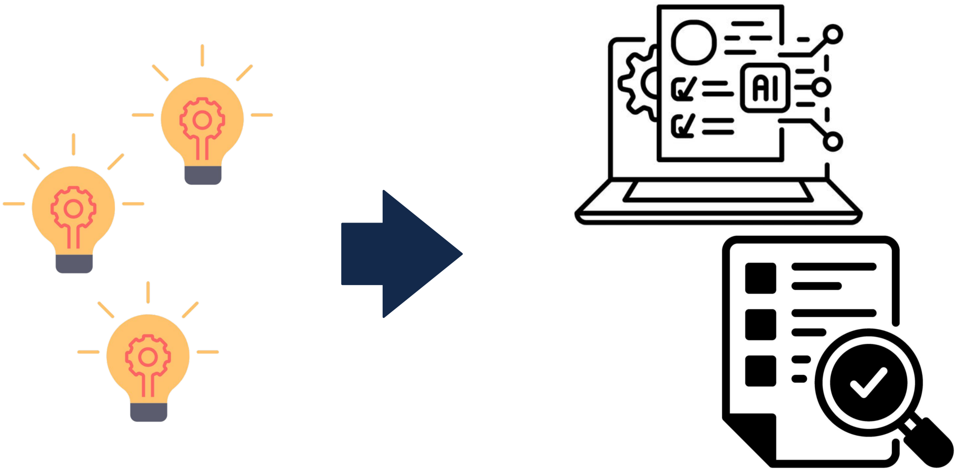
Artifact Evaluation: Enabling Reproducible Research

Bogdan "Bo" Stoica (UIUC)

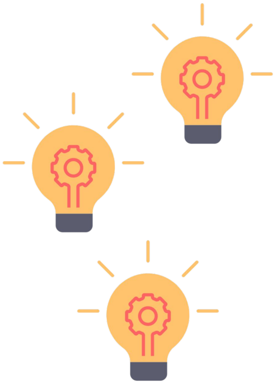
A typical “workflow” (Systems research)

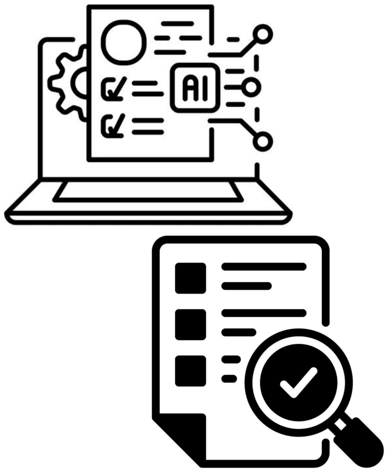


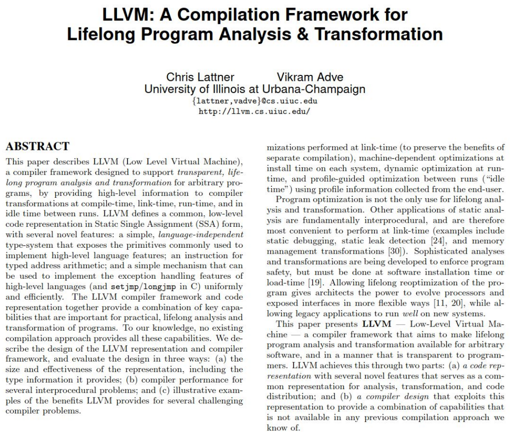
A typical “workflow” (Systems research)



A typical “workflow” (Systems research)







LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

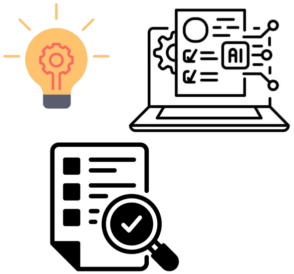
This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at run-time, and profile-guided optimization between runs (“idle time”) using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

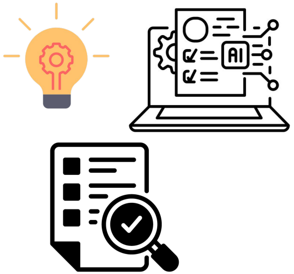
This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

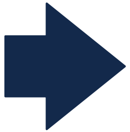
The peer-review process …



Paper preparation

The peer-review process ...





LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

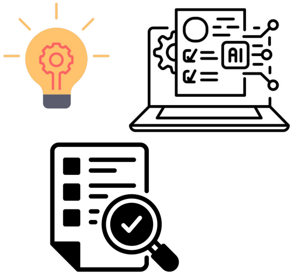
Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

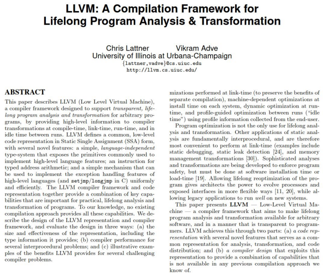
This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

Paper preparation

Paper deadline

The peer-review process …





LLVM: A Compilation Framework for
Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

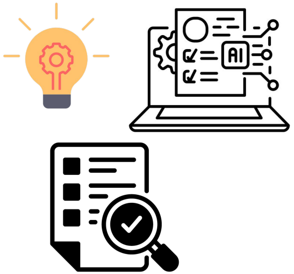
This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

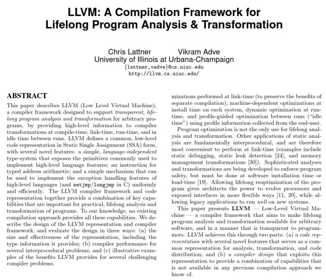
Peer Reviewing …

Paper preparation

Paper deadline

The peer-review process ...





LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

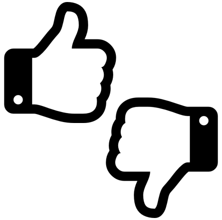
ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

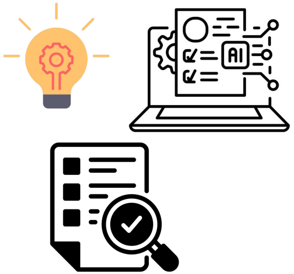
This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

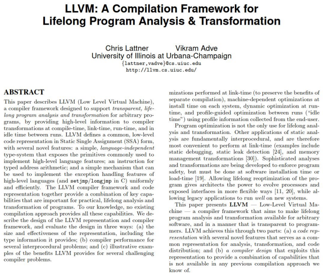


Paper preparation | **Paper deadline** | **Paper notification**

The peer-review process …

Artifact Evaluation





LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

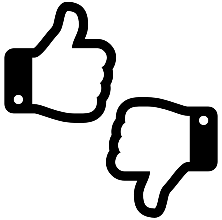
ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.



Paper preparation | **Paper deadline** | **Paper notification**

Artifact evaluation: a definition …

Artifact evaluation (AE) is the process of verifying that the artifacts released alongside a research paper (source code, datasets, scripts, configuration, etc.) faithfully correspond to the paper's description, and that they can be used to reproduce (or at least substantiate) the core paper claims, experimental setup, and reported results.

A bit of history ...

Software Engineering

FSE'11**

*** SIGMOD ran a similar initiative 2008-2011, but a formal AE process started with at FSE'11*

A bit of history ...

Software Engineering	Programming Language	Security & Privacy	Systems
FSE'11**	**OOPSLA'13**	**WiSec'17**	**SOSP'19**

*** SIGMOD ran a similar initiative 2008-2011, but a formal AE process started with at FSE'11*

A bit of history ...

Software Engineering	Programming Language	Security & Privacy	Systems
FSE'11**	**OOPSLA'13**	**WiSec'17**	**SOSP'19**
ASE	ASPLOS	Security	OSDI
ICSE	PLDI	S&P	EuroSys
ISSTA	POPL	CSS	FAST
MSR	PPoPP	NDSS	SIGCOMM
...	...	...	NSDI
			...

*** SIGMOD ran a similar initiative 2008-2011, but a formal AE process started with at FSE'11*

A bit of history ...

Software Engineering	Programming Language	Security & Privacy	Systems	Computer Architecture
FSE'11**	**OOPSLA'13**	**WiSec'17**	**SOSP'19**	**AI/ML**
ASE	ASPLOS	Security	OSDI	**Database Systems**
ICSE	PLDI	S&P	EuroSys	**Cryptography**
ISSTA	POPL	CSS	FAST	...
MSR	PPoPP	NDSS	SIGCOMM	
...	...	...	NSDI	
			...	

*** SIGMOD ran a similar initiative 2008-2011, but a formal AE process started with at FSE'11*

Artifact evaluation: goals

Increase thrust of published research (artifact “badges”)

Artifact evaluation: goals

Increase thrust of published research (artifact “badges”)

Ensure artifacts are available & easily accessible

Artifact evaluation: goals

Increase thrust of published research (artifact "badges")

Ensure artifacts are available & easily accessible

Facilitate reproducibility of key findings

Artifact evaluation: goals

Increase thrust of published research (artifact “badges”)

Ensure artifacts are available & easily accessible

Facilitate reproducibility of key findings

Enable reusability & extensibility

Artifact evaluation: badges

Artifact Available: publicly & permanently available

Artifact evaluation: badges

Artifact Available: publicly & permanently available

Artifact Functional: documented, exercisable, and includes validation

Artifact Reusable: repurposable, modular, and extensible

Artifact evaluation: badges

Artifact Available: publicly & permanently available

Artifact Functional: documented, exercisable, and includes validation

Artifact Reusable: repurposable, modular, and extensible

Result Reproduced: re-obtained by using, in part, author-provided artifacts

Result Replicated: re-obtained without author-provided artifacts

Artifact evaluation: badges (Systems)

Artifact Available: publicly & permanently available

Artifact Functional: documented, exercisable, and includes validation

Result Reproduced: re-obtained by using, in part, author-provided artifacts

Artifact evaluation timeline

Preparation	Evaluation

Artifact evaluation timeline

Paper notification

Preparation

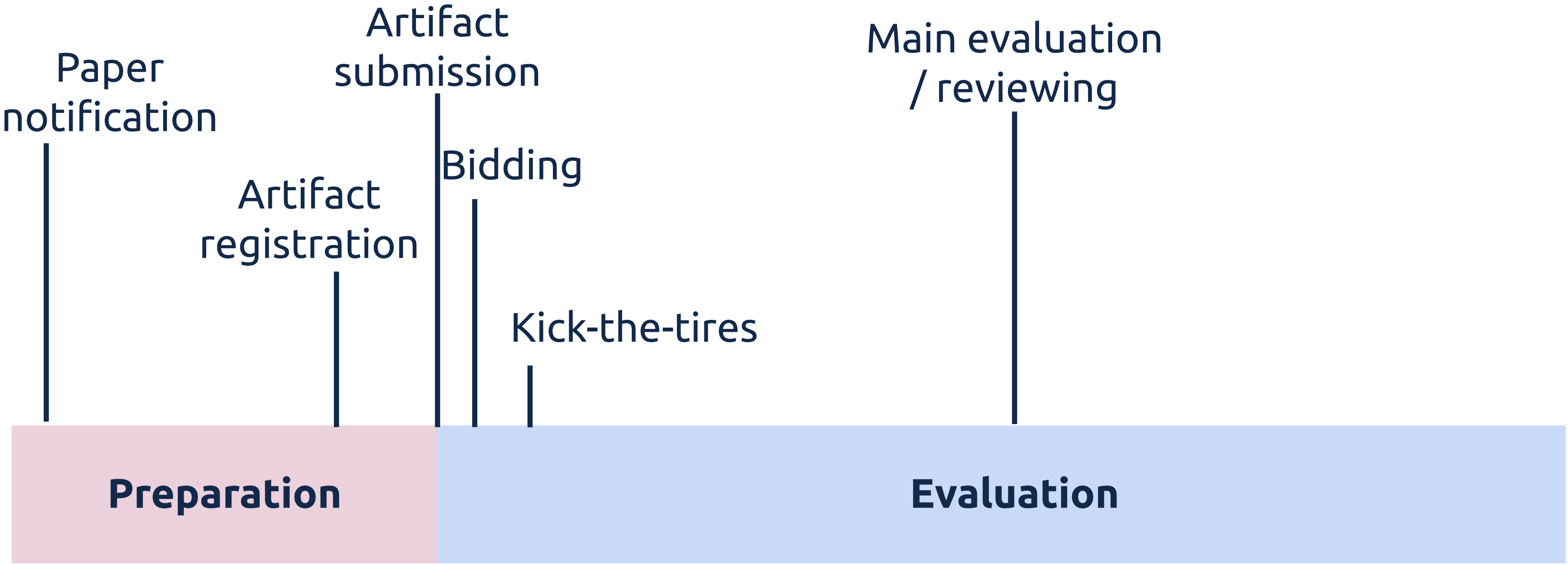
Evaluation

Artifact evaluation timeline

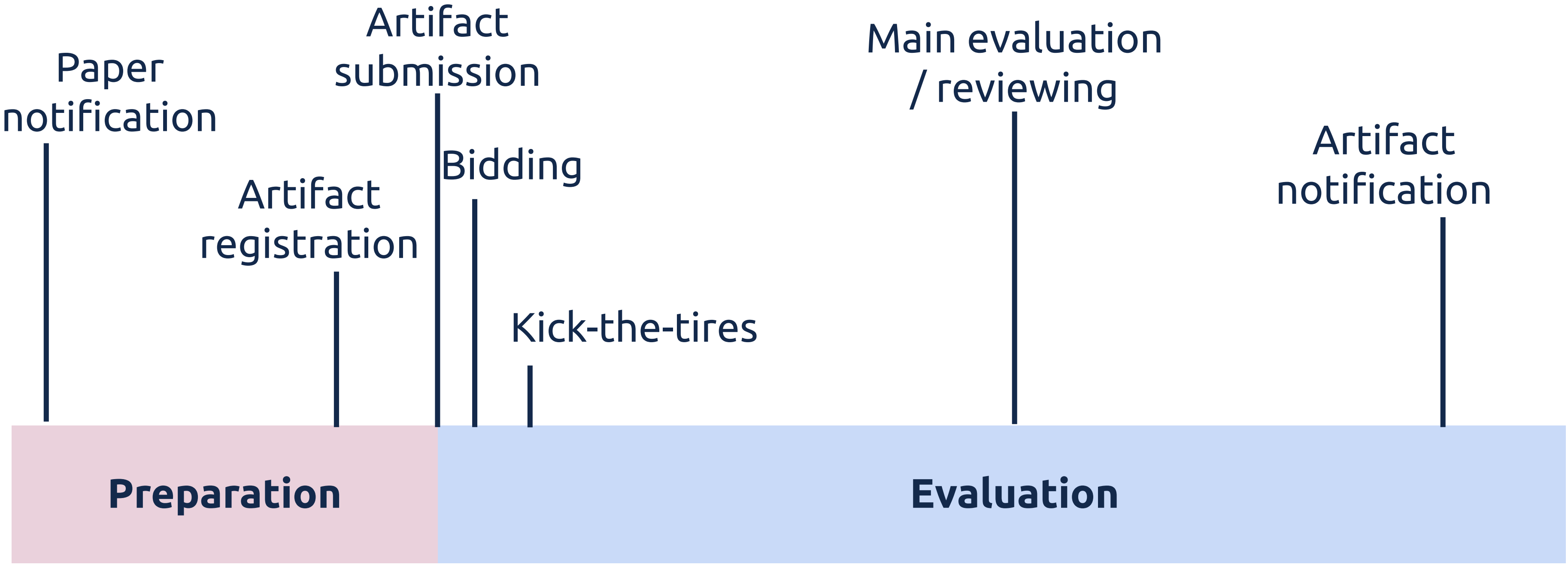
Artifact evaluation timeline

Artifact evaluation timeline

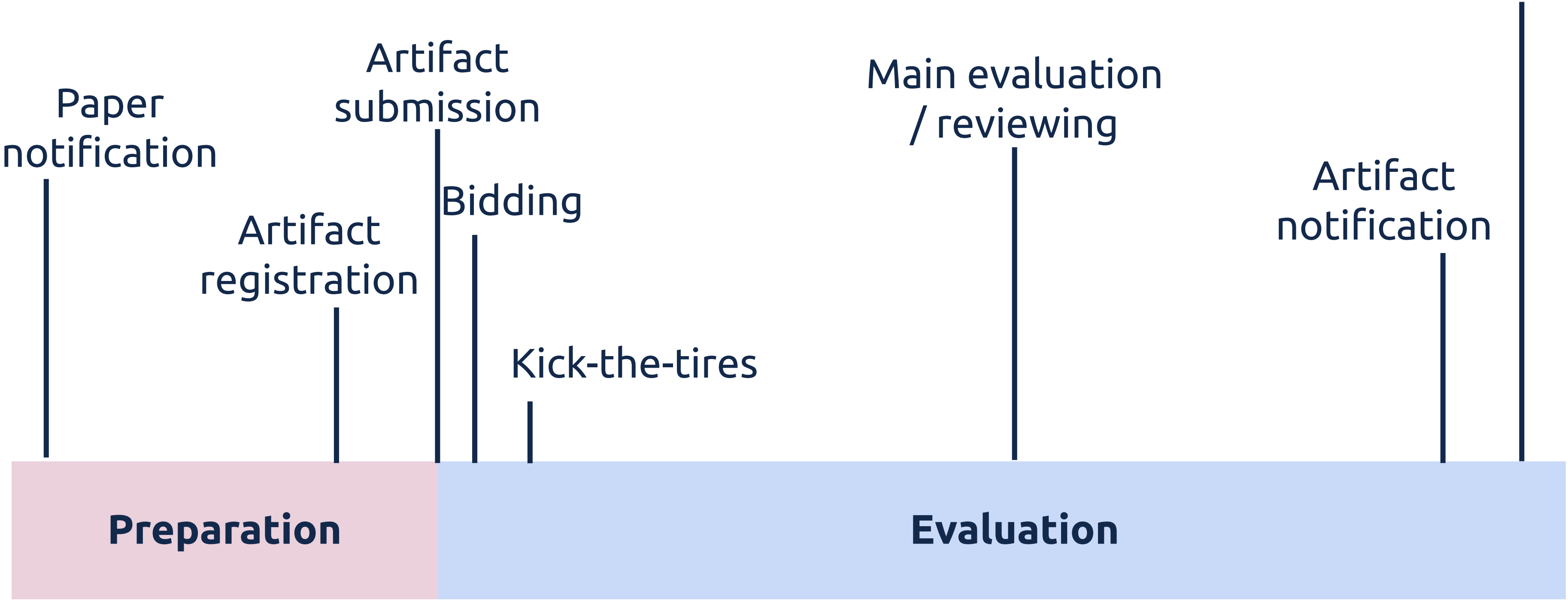
Artifact evaluation timeline



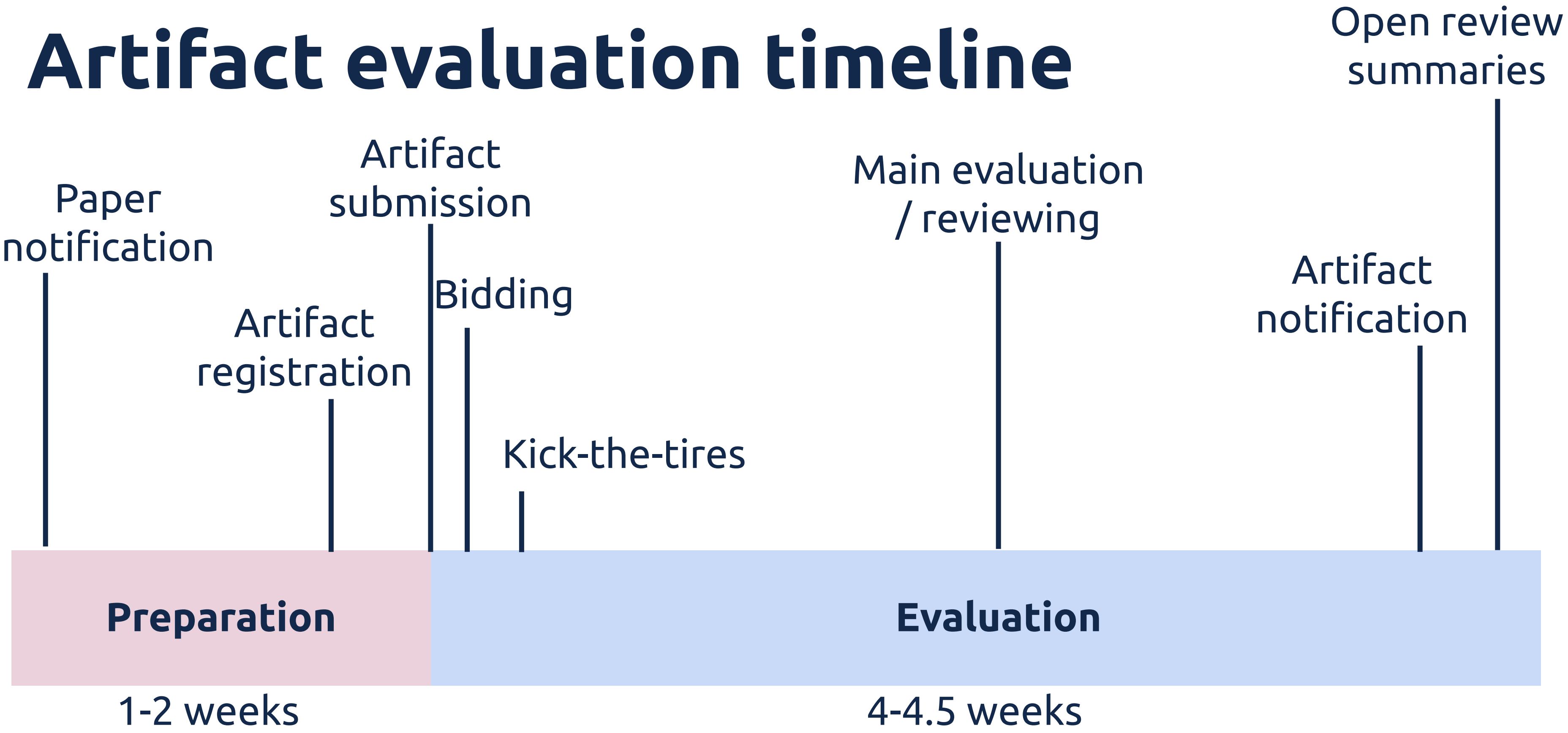
Artifact evaluation timeline



Artifact evaluation timeline



Artifact evaluation timeline



The role of authors

Prepare a self-contained artifact w/ persistent hosting

The role of authors

Prepare a self-contained artifact w/ persistent hosting

Write clear guidelines (appendix and/or README)

The role of authors

Prepare a self-contained artifact w/ persistent hosting

Write clear guidelines (appendix and/or README)

Provide a minimal, simple experiment as “running example”

The role of authors

Prepare a self-contained artifact w/ persistent hosting

Write clear guidelines (appendix and/or README)

Provide a minimal, simple experiment as “running example”

Engage with reviewers to improve the artifact

The role of reviewers

Evaluate the artifact, but also audit paper-code alignment

The role of reviewers

Evaluate the artifact, but also audit paper-code alignment

Start early, engage with the authors

The role of reviewers

Evaluate the artifact, but also audit paper-code alignment

Start early, engage with the authors

Follow the Chairs’ guidelines and provided badge checklist

The role of reviewers

Evaluate the artifact, but also audit paper-code alignment

Start early, engage with the authors

Follow the Chairs' guidelines and provided badge checklist

Write a thorough, detailed, and respectful review

Available infrastructure

Individual machines (e.g., desktop, laptop)

Available infrastructure

Individual machines (e.g., desktop, laptop)

Academic cloud infrastructure

CloudLab

ChameleonCloud

Available infrastructure

Individual machines (e.g., desktop, laptop)

Academic cloud infrastructure

CloudLab

ChameleonCloud

Commercial cloud infrastructure

aws

Azure

Available infrastructure

Individual machines (e.g., desktop, laptop)

CloudLab

Academic cloud infrastructure

ChameleonCloud

aws

Commercial cloud infrastructure

Azure

Most frequent challenges

C1: Short preparation & review windows

Most frequent challenges

C1: Short preparation & review windows

C2: Persistent artifact availability

Most frequent challenges

C1: Short preparation & review windows

C2: Persistent artifact availability

C3: Specialized hardware requirements

Most frequent challenges

C1: Short preparation & review windows

C2: Persistent artifact availability

C3: Specialized hardware requirements

C4: Environment setup, configuration, and installation friction

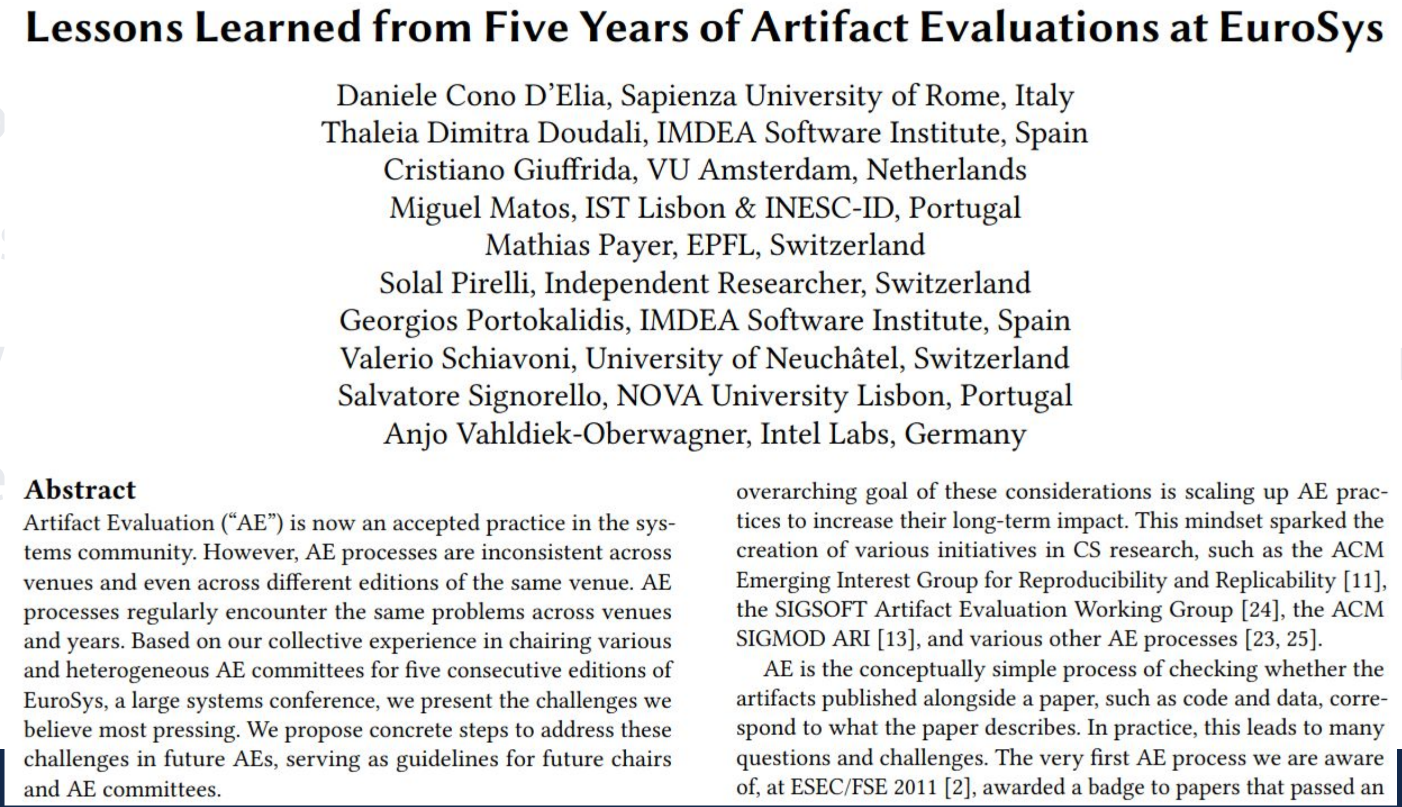
Most frequent challenges

C1: Sho

C2: Per

C3: Env … riction

C4: Spe



Lessons Learned from Five Years of Artifact Evaluations at EuroSys

Daniele Cono D'Elia, Sapienza University of Rome, Italy
Thaleia Dimitra Doudali, IMDEA Software Institute, Spain
Cristiano Giuffrida, VU Amsterdam, Netherlands
Miguel Matos, IST Lisbon & INESC-ID, Portugal
Mathias Payer, EPFL, Switzerland
Solal Pirelli, Independent Researcher, Switzerland
Georgios Portokalidis, IMDEA Software Institute, Spain
Valerio Schiavoni, University of Neuchâtel, Switzerland
Salvatore Signorello, NOVA University Lisbon, Portugal
Anjo Vahldiek-Oberwagner, Intel Labs, Germany

Abstract

Artifact Evaluation ("AE") is now an accepted practice in the systems community. However, AE processes are inconsistent across venues and even across different editions of the same venue. AE processes regularly encounter the same problems across venues and years. Based on our collective experience in chairing various and heterogeneous AE committees for five consecutive editions of EuroSys, a large systems conference, we present the challenges we believe most pressing. We propose concrete steps to address these challenges in future AEs, serving as guidelines for future chairs and AE committees.

overarching goal of these considerations is scaling up AE practices to increase their long-term impact. This mindset sparked the creation of various initiatives in CS research, such as the ACM Emerging Interest Group for Reproducibility and Replicability [11], the SIGSOFT Artifact Evaluation Working Group [24], the ACM SIGMOD ARI [13], and various other AE processes [23, 25].

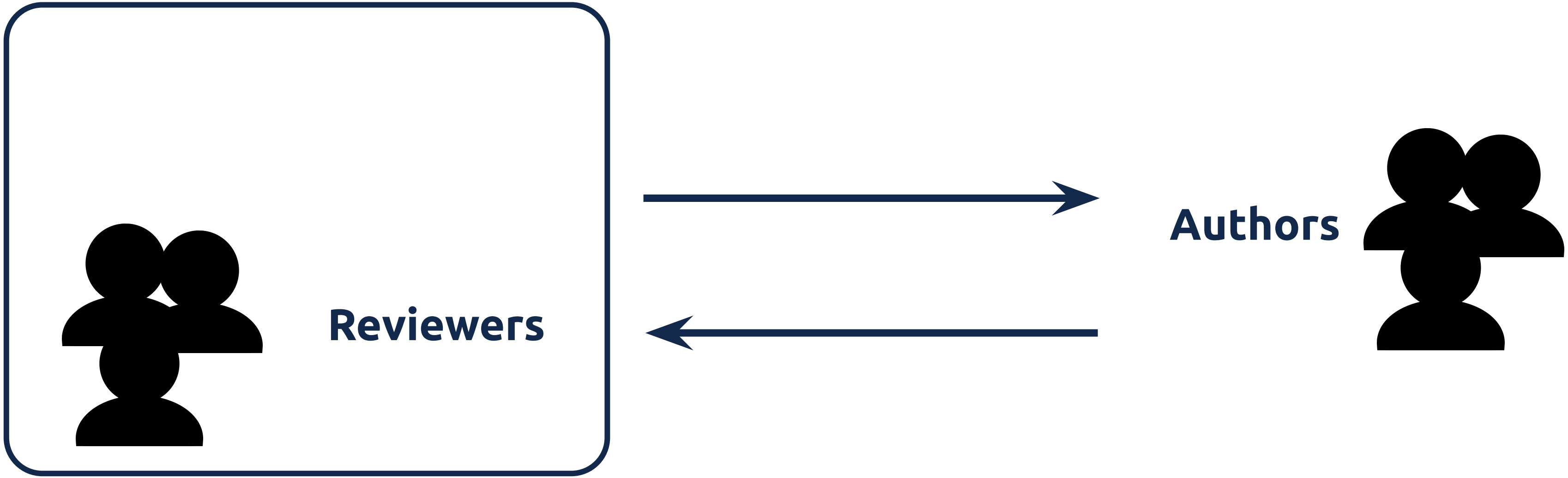
AE is the conceptually simple process of checking whether the artifacts published alongside a paper, such as code and data, correspond to what the paper describes. In practice, this leads to many questions and challenges. The very first AE process we are aware of, at ESEC/FSE 2011 [2], awarded a badge to papers that passed an

Future of Artifact Evaluation ...

C4: Environment setup, configuration, and installation friction

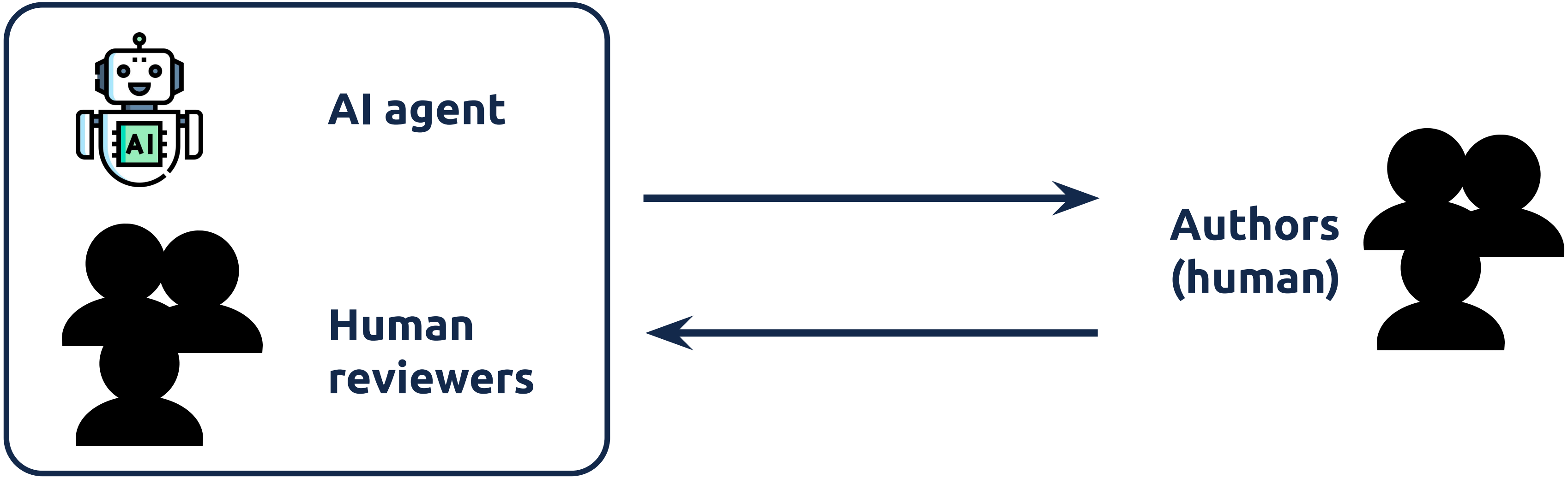
Future of Artifact Evaluation …

C4: Environment setup, configuration, and installation friction



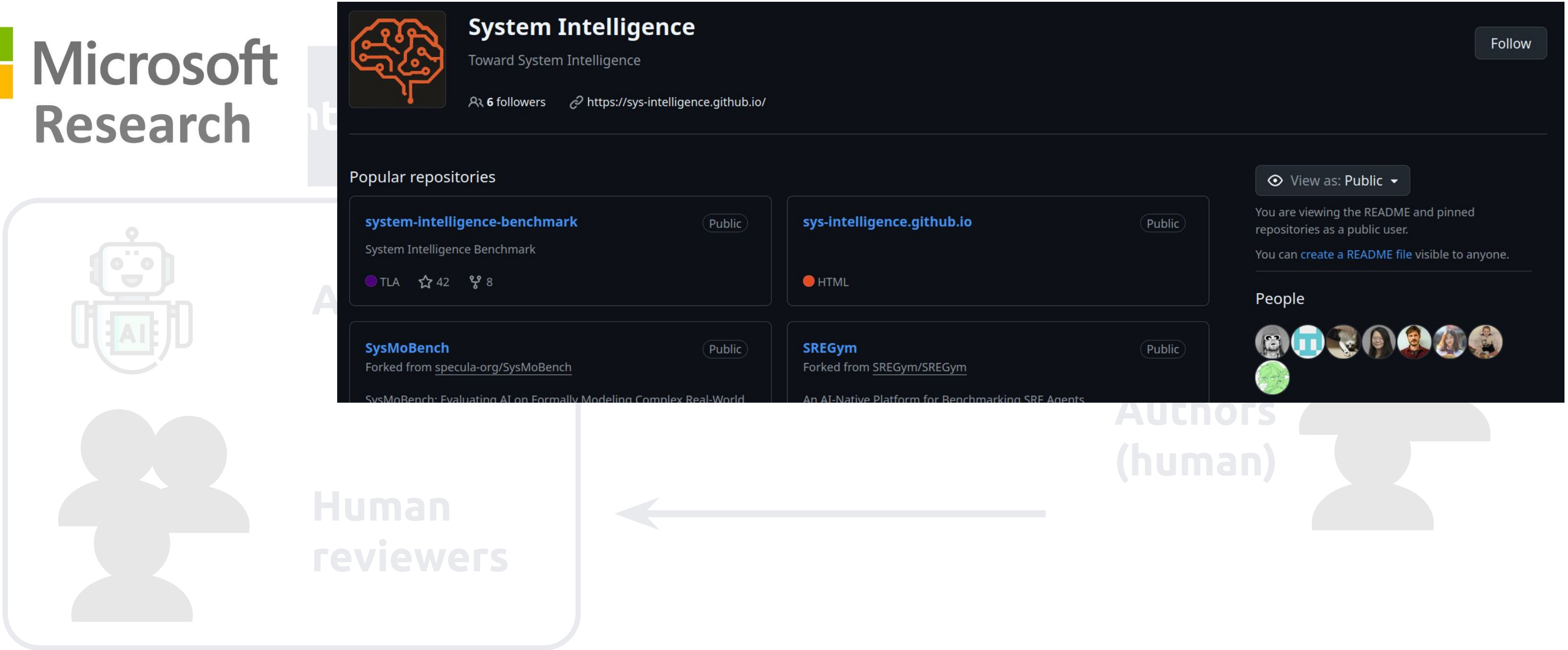
Future of Artifact Evaluation …

C4: Environment setup, configuration, and installation friction



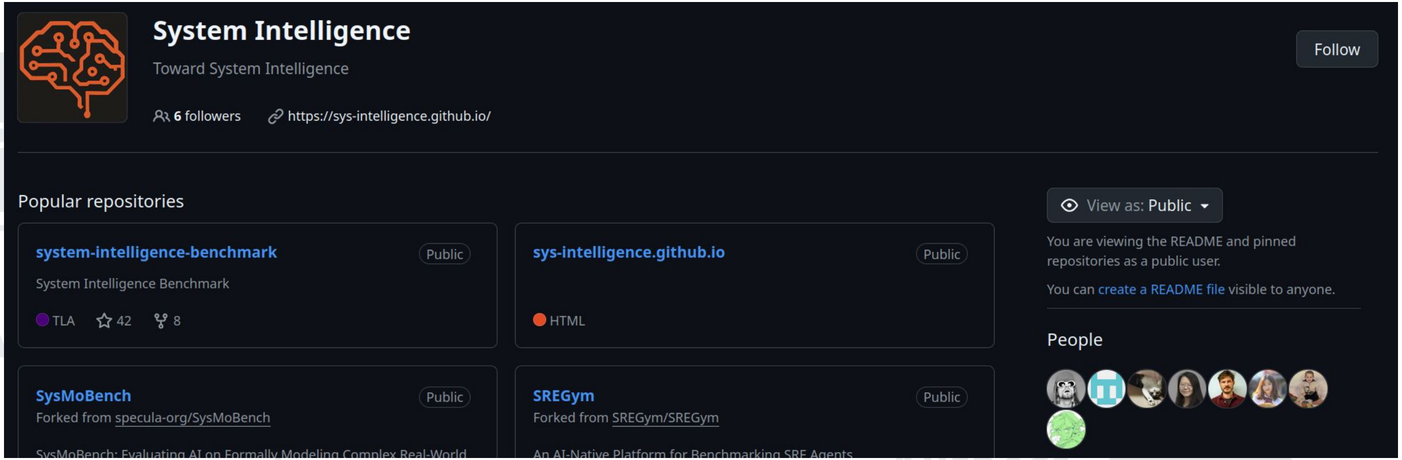
Future of Artifact Evaluation …

Microsoft Research



Future of Artifact Evaluation …

Microsoft Research



Check out the "System Intelligence" series on ACM SIGOPS Blog

Want to get involved?

Ongoing: EuroSys'26 (email us by Jan 30: aec-2026@eurosys.org)

Upcoming: OSDI'26, SOSP'26, EuroSys'27, ASPLOS'27, etc.

Want to get involved?

Ongoing: EuroSys'26 (email us by Jan 30: aec-2026@eurosys.org)

Upcoming: OSDI'26, SOSP'26, EuroSys'27, ASPLOS'27, etc.

SysIntellignce: -- contact Bo (bastoica@illinois.edu)
-- drop by @ https://github.com/sys-intelligence/

Thank you!