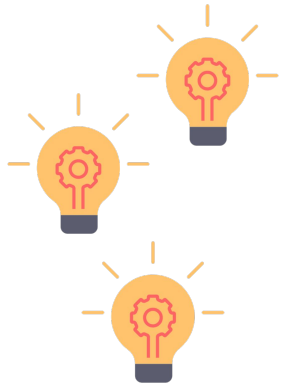


Artifact Evaluation: Enabling Reproducible Research

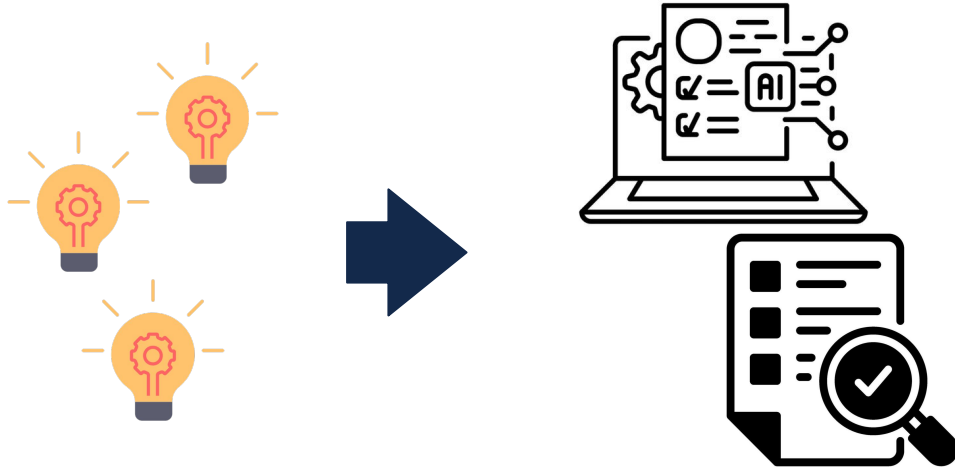
Bogdan “Bo” Stoica (UIUC)



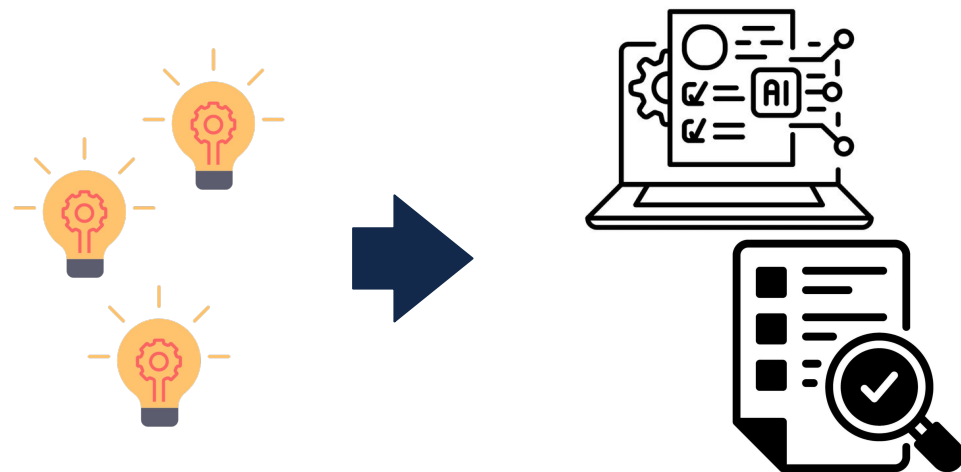
A typical “workflow” (Systems research)



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

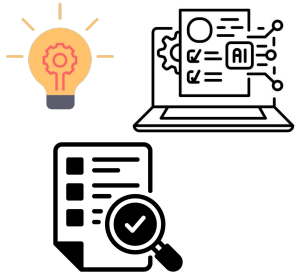
This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between 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 link 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**



Paper deadline

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
(lattner,adve)@cs.uiuc.edu

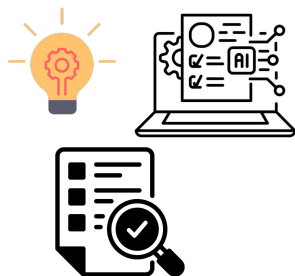
[illegible]

optimizations performed at link-time (so preserve the benefits of separate compilation), middle-code based optimizations at install time on each system, dynamic optimization at run-time, and profile-guided optimization between runs ("side-tuning").

Program optimization is not the only use for linking and run-time transformation. Other applications of static analysis include: (a) detecting errors in program code, (b) detecting errors in program data, (c) detecting errors in program state, and (d) detecting errors in program transformations not being done to achieve program safety, but that *should* be done before installation time or side-tune [29]. Allowing linking reorganization of the program is also useful for (e) detecting errors in the program's exposed interfaces in more flexible ways [31, 26], while allowing legacy applications to run on new systems.

Link-time transformation is a new paradigm for static analysis, a compiler framework that aims to make SSA program analysis and transformation available for arbitrary programs, not just those that can be compiled by a compiler. LLVM achieves this through two parts: (a) a new representation with several novel features that serves as a common representation for analysis, transformation, and code generation, and (b) a set of tools that use this representation to provide a contribution of capabilities that is not available in any previous compilation approach.

The peer-review process ...



Paper
preparation



LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner, adve}@illinois.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 lightweight mechanisms to compile transformations at compile-time, link-time, run-time, and in the field, LLVM allows a compiler, the host code, or the user to perform transformations at any time. LLVM also provides a rich set of interfaces for extending the framework to support new transformations. The LLVM compiler framework and code generation infrastructure provide a foundation for many other tools that are important to program analysis and transformation of programs. It can be used to implement the design of the LLVM representation and compiler framework, and to build tools that are important to the analysis and transformation of programs. It can be used to implement the design of the LLVM representation and compiler framework, and to build tools that are important to the analysis and transformation of programs. It can be used to implement the design of the LLVM representation and compiler framework, and to build tools that are important to the analysis and transformation of programs.

minimization performed at link-time (to preserve the benefits of compile-time compilation), machine-dependent optimization at run-time (to take advantage of hardware optimization at run-time), and post-optimization optimization between runs ("side-step") using profile information collected from the end user. Program optimization is not the only use for linking and run-time transformation. Other applications of LLVM and LLVM-based tools include: (1) code generation and code transformation to perform at link-time (examples include static linking, static code downloading [16], and memory management transformation [16]). Sophisticated analysis and transformation are being developed to enhance program safety, but must be done at software installation (link or load-time) [16]. Allowing linking representation of the program gives developers the power to create platform and system interfaces in more flexible ways [14, 16], while allowing binary applications to run on new systems.

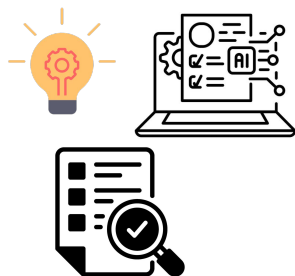
This paper presents LLVM - Low-Level Virtual Machine - a compiler framework that aims to make linking, analysis, and transformation of programs a simple and transparent task. LLVM is designed to be transparent to program analysis, LLVM allows this through transparent, link-time compilation with several special features that serve as a common representation for analysis, transformation, and code distribution, and (3) a compiler design that requires the programmer to provide a representation of compilation that is not available in any previous compilation approach we know of.

Peer
Reviewing ...



Paper
deadline

The peer-review process ...



LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

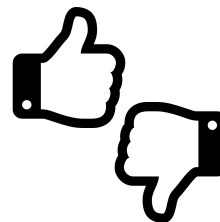
Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner, adve}@illinois.edu
<http://llvm.cs.uiuc.edu/>

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 lightweight mechanisms to compute transformations at compile-time, link-time, run-time, and in the life-time between runs, LLVM defines a reusable, low-level code representation in Basic Block Intermediate Representation (BBIR), with several novel features: a simple, language-independent representation that supports the previous mentioned and to implement high-level language features, an interface for typed address arithmetic, and a simple mechanism that can be used to implement the exception handling features of high-level languages and enable programs in C to run safely and efficiently. The LLVM compiler framework and code representation together provide a foundation for many systems that are important to program analysis, program transformation, and program execution. In our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and outline the design in three major (a) the use and distribution of the representation, including the type information it provides (b) compiler performance for several representative problems, and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

minimize performance at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations are limited due to such errors. Machine optimization at run-time, and post-optimization optimization between runs ("side step") using profile information collected from the end user. Program optimization is not the only use for linking analysis and transformation. Other applications of analysis and transformation that require link-time information include static debugging, static code analysis [16], and memory management transformation [17]. Sophisticated analysis and transformation are being developed to reduce program safety, but must be done at software installation time or link-time [18]. Allowing linking representation of the program gives developers the power to create platform and targeted interfaces to more flexible ways [14, 19], while allowing higher applications to run on a new system.

This paper presents LLVM - Low-Level Virtual Machine - a compiler framework that aims to enable linking analysis and transformation for arbitrary programs, and in a manner that is transparent to programmers. LLVM achieves this through transparency, and a set of mechanisms 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.



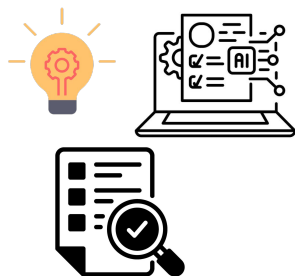
Paper
preparation

Paper
deadline

Paper
notification

The peer-review process ...

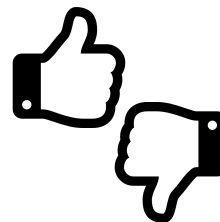
Artifact Evaluation



LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner, adve}@illinois.edu
<http://llvm.org/docs.html>

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 lightweight facilities to compute transformations at compile-time, link-time, run-time, and in the life-time between runs, LLVM defines a reusable, low-level code representation in that Single-Byte Instruction (SBI) form, with several novel features: a simple, language-independent representation that supports the previous mentioned and to implement high-level language features, an interface for typed address arithmetic, and a simple mechanism that can be used to implement the exception handling features of high-level languages and arbitrary programs in C, uniformly and efficiently. The LLVM compiler framework and code representation together provide a foundation for many systems that are important to program analysis, program transformation, and program execution. In our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and outline the design in more detail for the use and distribution of the representation, including the type information it provides. (i) compile performance for several representative problems, and (ii) illustrative examples of the benefits LLVM provides for several challenging compile problems.



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**

FSE'11**

**Programming
Language**

OOPSLA'13

**Security &
Privacy**

WiSec'17

Systems

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

FSE'11**

ASE
ICSE ISSTA
MSR
...

Programming Language

OOPSLA'13

ASPLOS
PLDI POPL
PPoPP
...

Security & Privacy

WiSec'17

Security
CSS S&P
NDSS
...

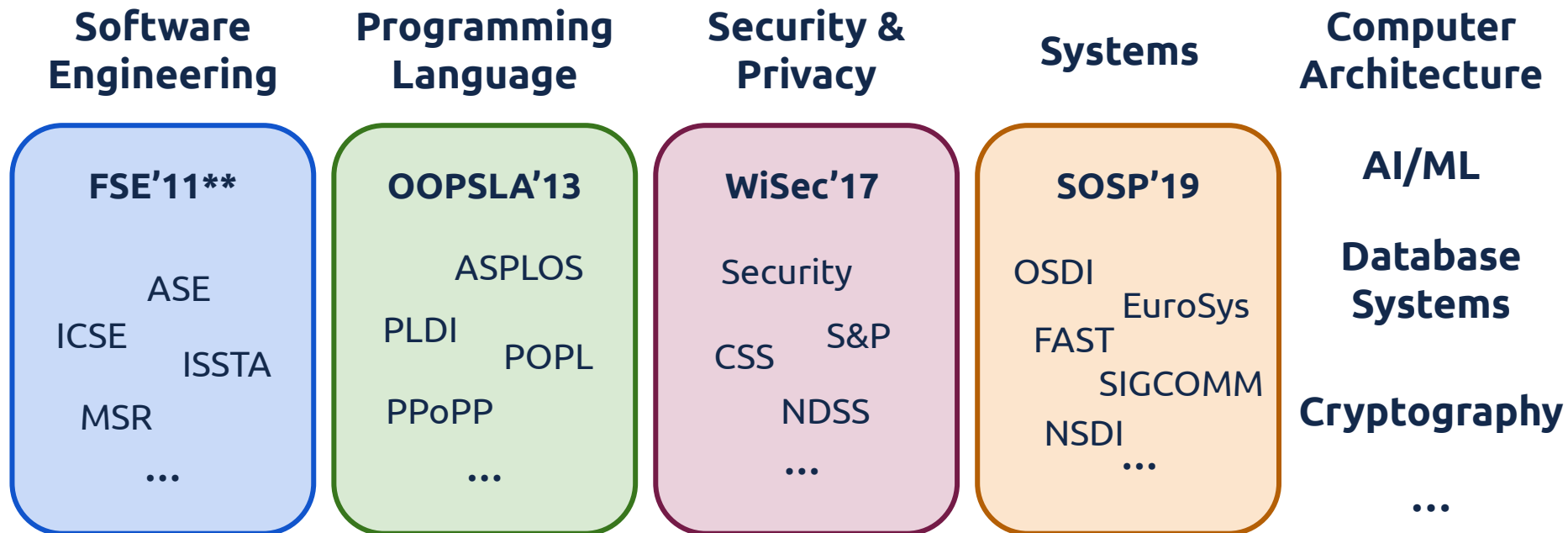
Systems

SOSP'19

OSDI EuroSys
FAST SIGCOMM
NSDI
...

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

A bit of history ...



*** 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



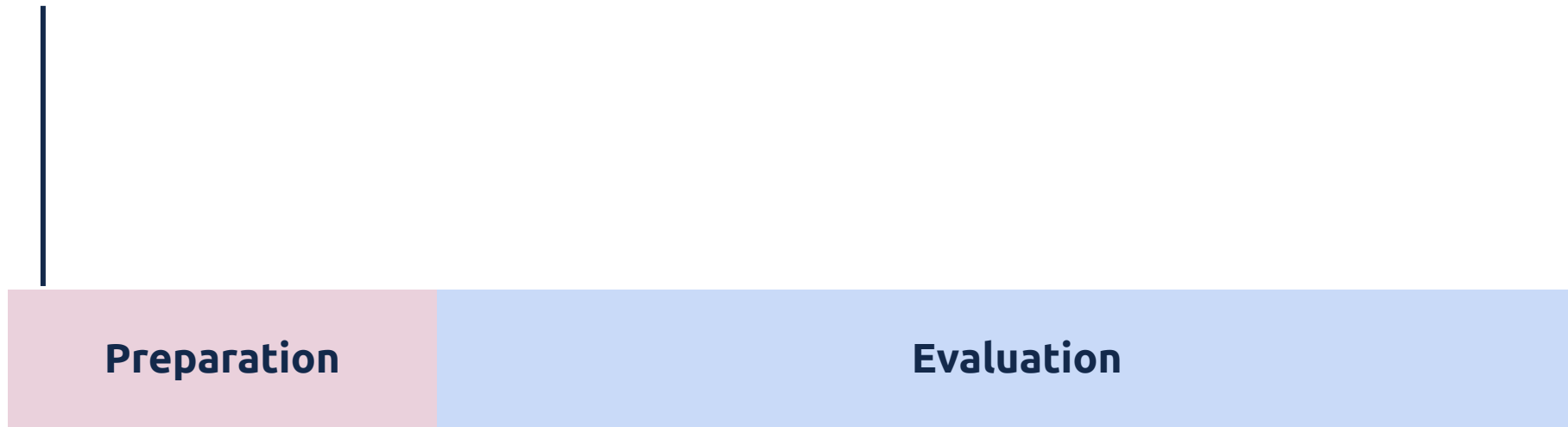
A horizontal timeline bar divided into two segments. The left segment is light red and labeled 'Preparation'. The right segment is light blue and labeled 'Evaluation'.

Preparation

Evaluation

Artifact evaluation timeline

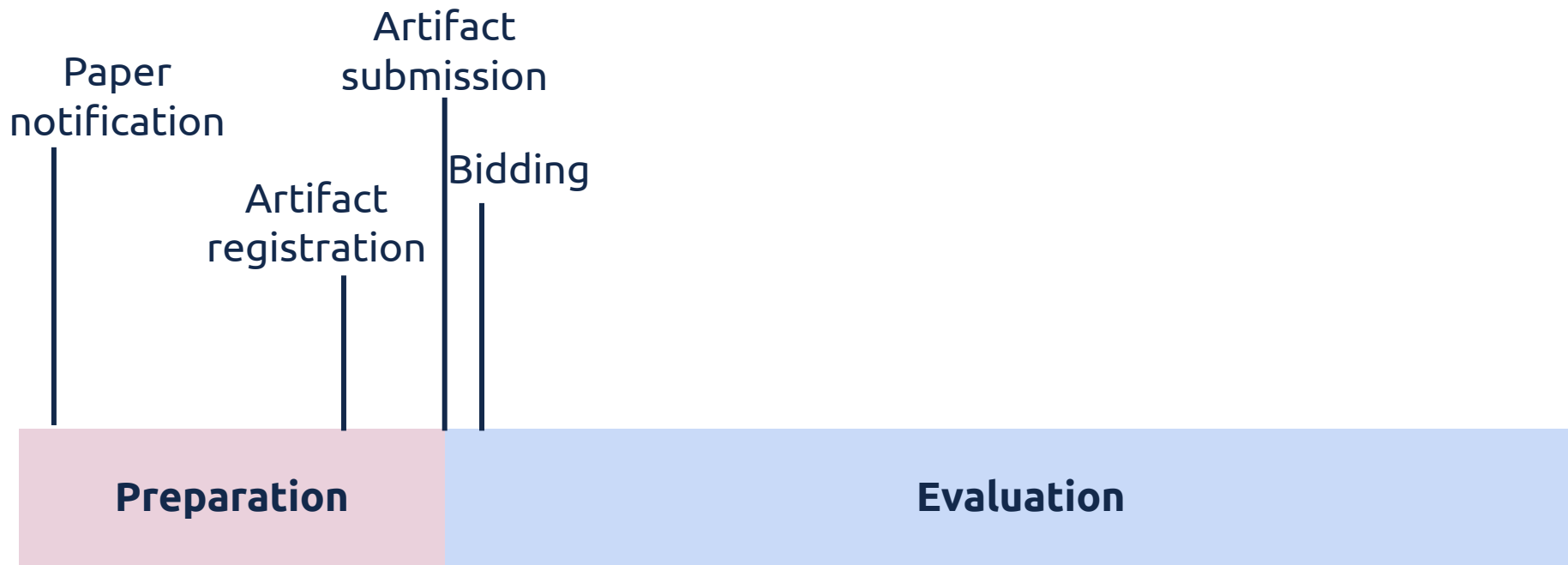
Paper
notification



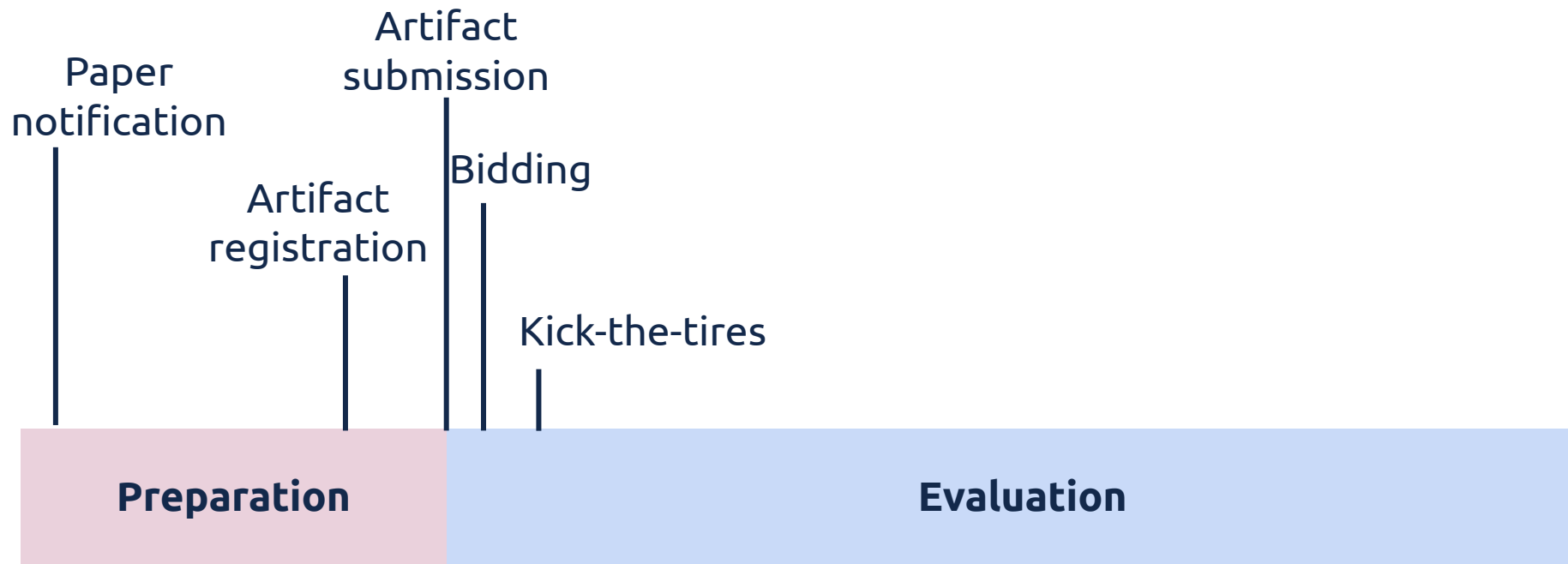
Artifact evaluation timeline



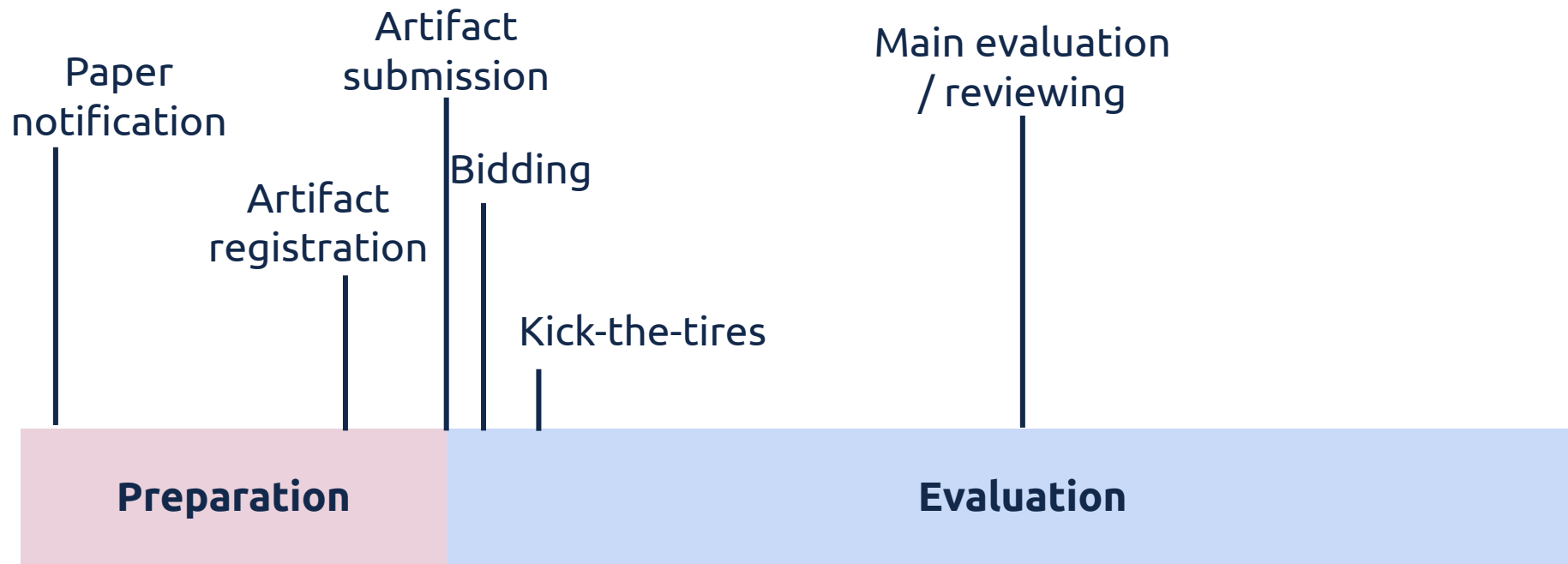
Artifact evaluation timeline



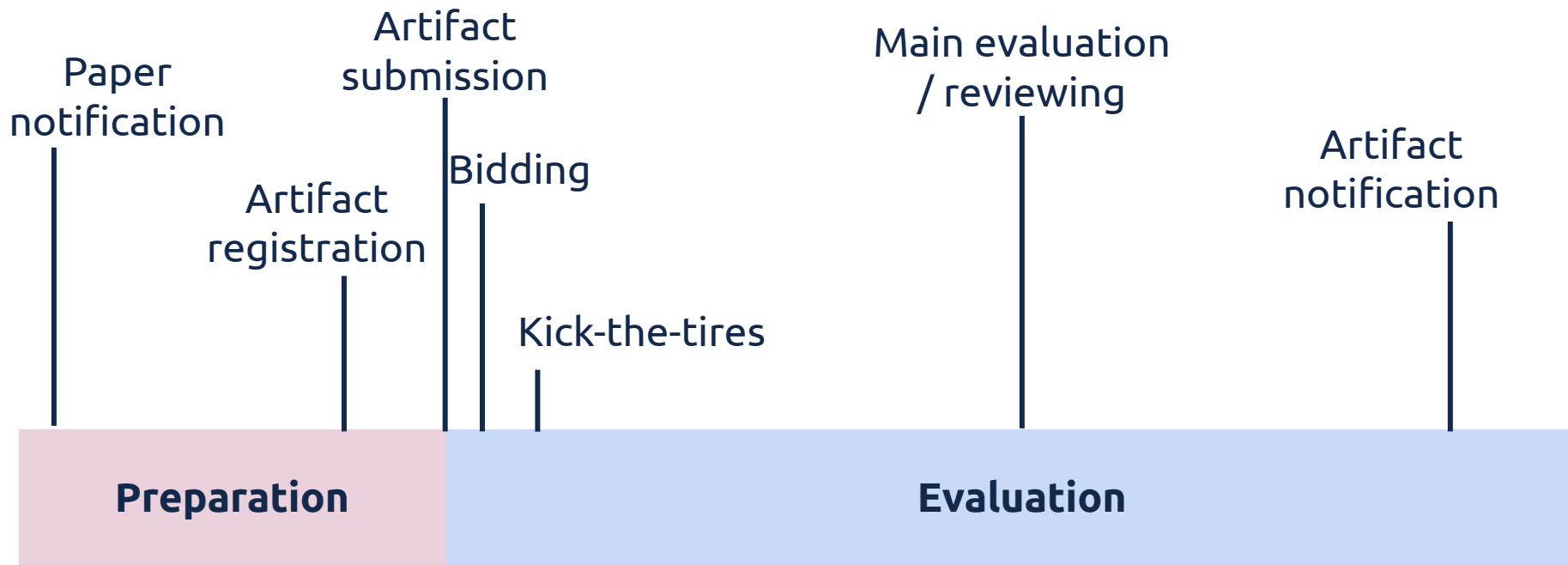
Artifact evaluation timeline



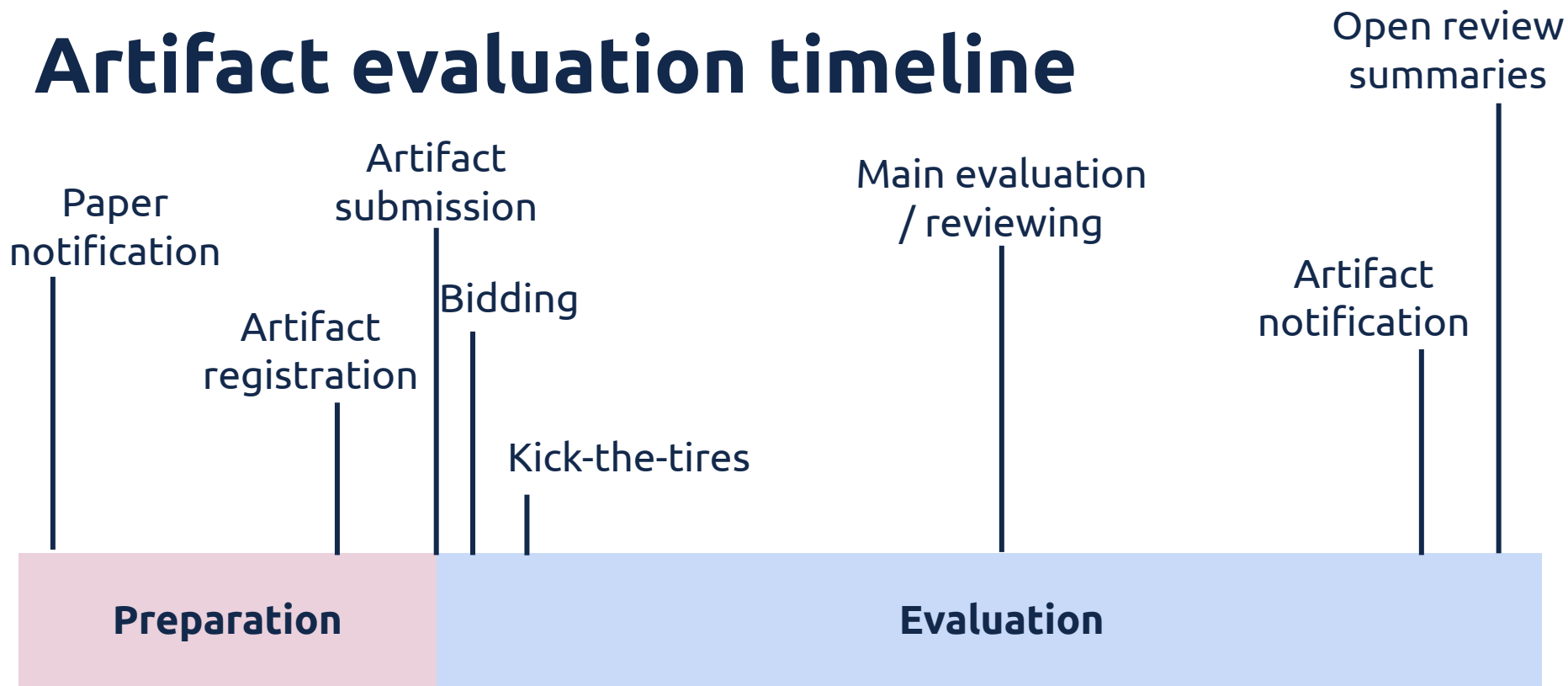
Artifact evaluation timeline



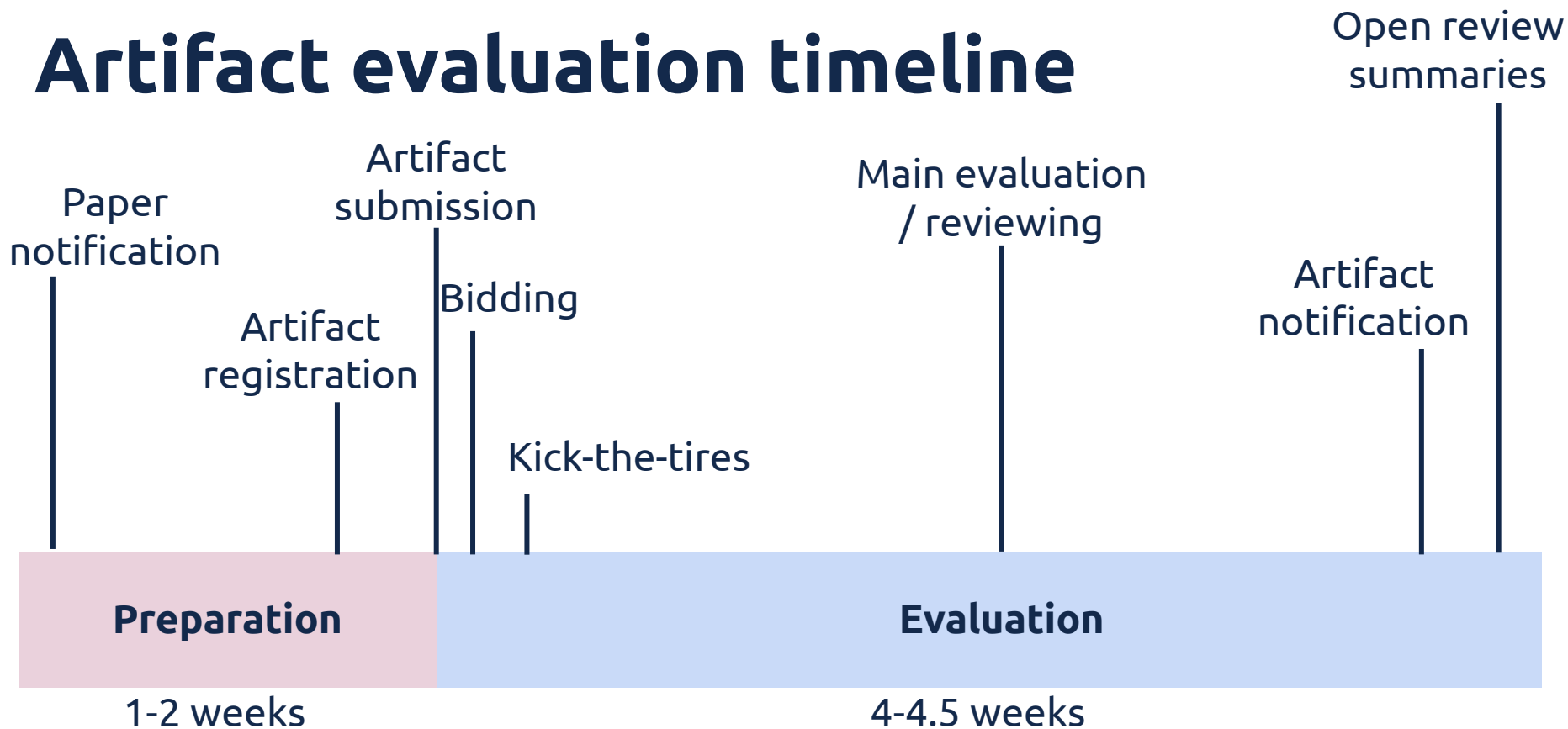
Artifact evaluation timeline



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



Available infrastructure

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

Academic cloud infrastructure

Commercial cloud infrastructure



Available infrastructure

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

Academic cloud infrastructure



Commercial cloud infrastructure



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

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].

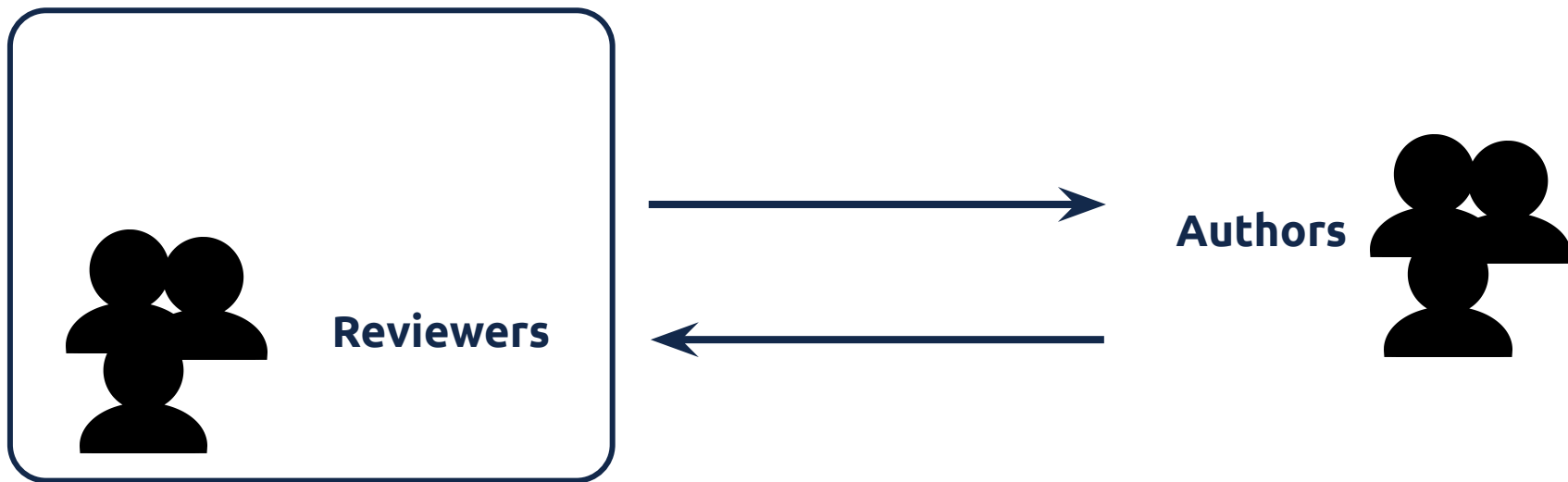
AE is the conceptually simple process of checking whether the artifacts published alongside a paper, such as code and data, correspond to what the paper describes. In practice, this leads to many questions and challenges. The very first AE process we are aware of, at ESEC/FSE 2011 [2], awarded a badge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

Future of Artifact Evaluation ...

C4: Environment setup, configuration, and installation friction

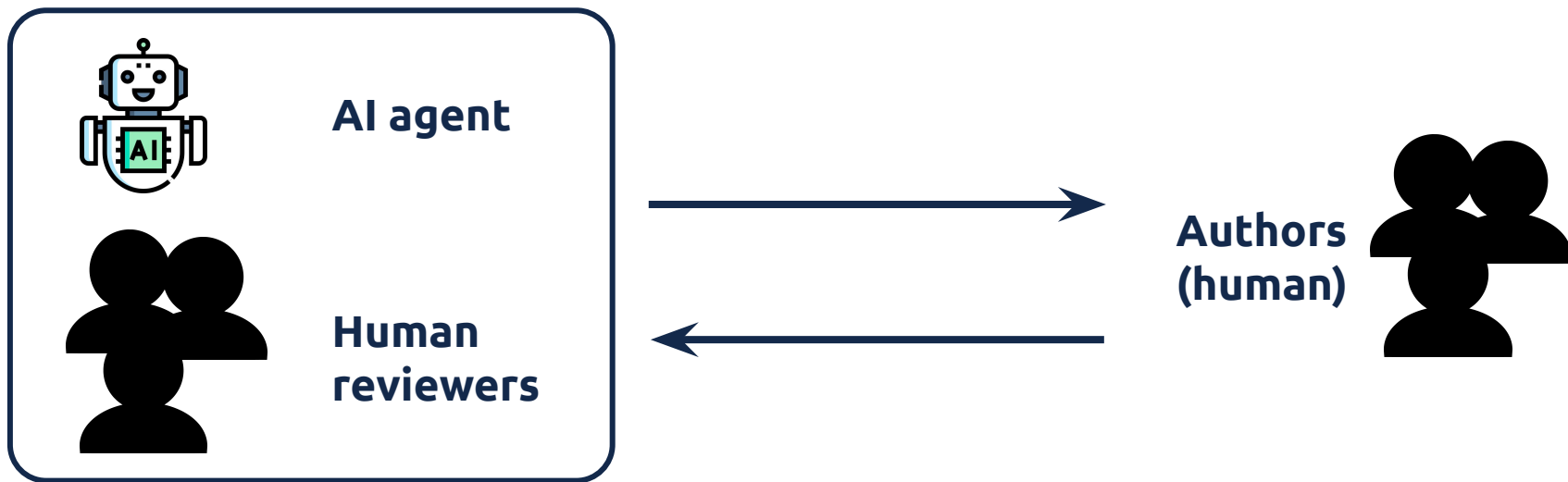
Future of Artifact Evaluation ...

C4: Environment setup, configuration, and installation friction



Future of Artifact Evaluation ...

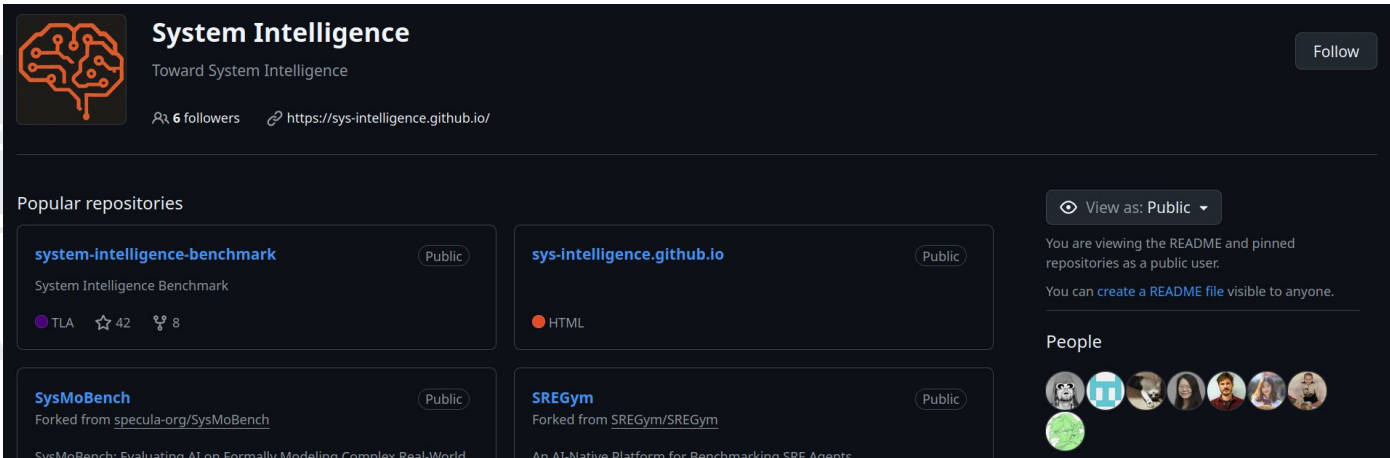
C4: Environment setup, configuration, and installation friction



Future of Artifact Evaluation ...



Microsoft
Research



System Intelligence
Toward System Intelligence
6 followers <https://sys-intelligence.github.io/> Follow

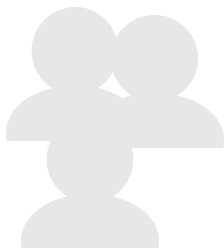
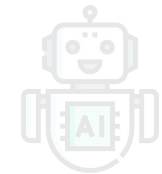
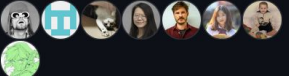
Popular repositories

| Repository | Stars | Language | Public |
|---|-------|----------|--------|
| system-intelligence-benchmark | 42 | TLA | Public |
| sys-intelligence.github.io | 8 | HTML | Public |
| SysMoBench | - | - | Public |
| SREGym | - | - | Public |

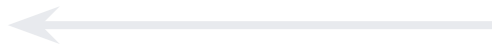
View as: Public

You are viewing the README and pinned repositories as a public user.
You can [create a README file](#) visible to anyone.

People



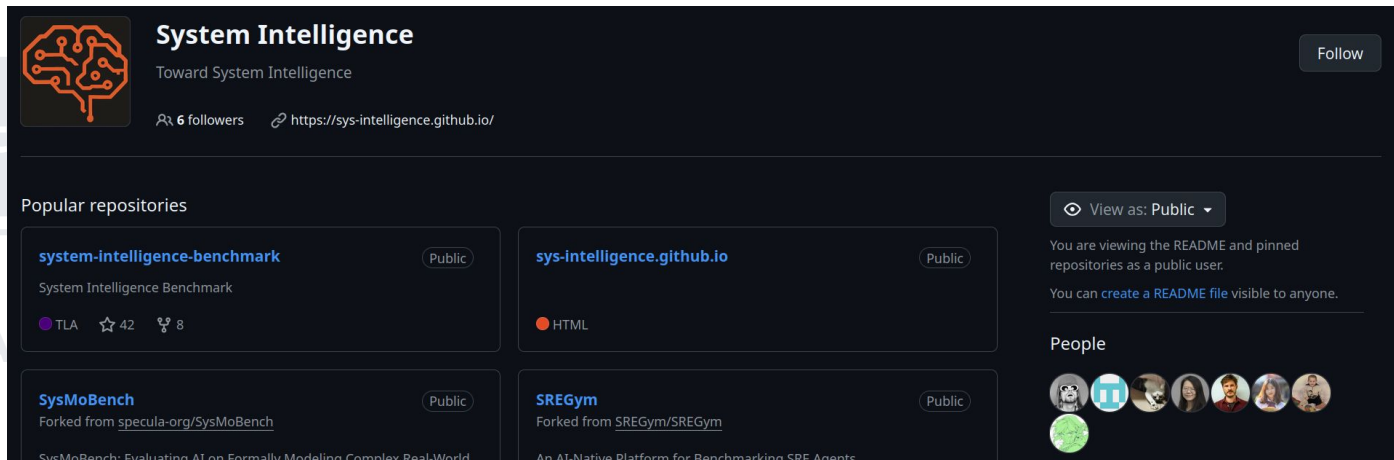
Human
reviewers



Authors
(human)



Future of Artifact Evaluation ...



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.

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

Thank you!