Research Statement

DAN PLYUKHIN

The world is full of distributed systems—from the networked devices that automate factories and businesses, to the geo-distributed data centers that serve AI models. Unfortunately, many distributed systems are *ticking time bombs* because of simple programming mistakes. For example, in 2015 a popular database called Cassandra gained a feature that involves requesting data multiple times; it wasn't until 2018 that developers realized the feature caused permanent data loss if the server happened to run a cleanup procedure between requests. Problems like this one are called *distributed concurrency bugs* (*DC bugs*): software problems that only manifest when events in a distributed system happen in a specific order. Since large systems have too many events to consider, catching DC bugs is like finding needles in an astronomically large haystack. The problem is only exacerbated by AI coding assistants, which are prone to writing buggy code. But what if we could guarantee our haystack doesn't contain needles in the first place?

I'm a **programming languages** and **distributed systems** researcher. My goal is to make distributed systems simpler, safer, and faster using insights from the theory of programming languages. For example:

1. Many DC bugs emerge when one server cleans up a resource that another server was still using. What if we had a *garbage collector* that safely cleans up distributed resources for us?
2. Many DC bugs emerge when the network delivers data in an unexpected order. What if we had a *compiler* generate each server's networking code, guaranteeing that all orderings will be handled properly?

These tools would be easy to implement if we constrained distributed systems to behave like sequential machines—but then execution time would skyrocket. To keep systems fast and scalable, we need to reimagine our tools in a setting where any device can crash and the network can't be trusted to deliver messages reliably.

To make this hard problem tractable, I build tools on top of powerful **abstractions** like the *actor model* and *choreographic programming*. Abstractions are useful because they make programmers give up some control in exchange for a simpler programming experience with stronger properties. For example: although garbage collection is computationally expensive in general distributed systems, I discovered it becomes a strictly easier sub-problem when we use the actor model. In the next section, I'll explain how I used this insight about distributed algorithms to develop **UIGC**, the first actor garbage collection framework that can clean up after crashed machines. I'll go on to show how my collaborators and I are using the theory of choreographic programming to develop the **Choral** compiler: an evolving toolkit for writing faster, safer, simpler distributed systems. At the end, I'll show how these contributions are coming together to build the foundations for next-generation distributed computing.

1 GARBAGE COLLECTION IN ACTOR SYSTEMS

Actors are lightweight processes that communicate by sending messages. For programmers, actors are useful because they encourage loose coupling and they make it easier to write highly concurrent software. For researchers, actors offer strong guarantees that we can harness, like the absence of shared state and low-level data races. Applications like Discord and WhatsApp use actors heavily for concurrency and fault-tolerance. Many organizations develop software inspired by actors, like Microsoft (in Orleans and Dapr), Apple (in Swift and FoundationDB), and Apache (in Pekko and YARN). **Billions** of people use actor systems every day—but there's a catch. Actors need to be killed when the application is finished with them, and traditional garbage collectors can't do the job for us. In fact:

> *In all four of the most popular distributed actor frameworks—**Pekko**, **Akka**, **Erlang**, and **Elixir**—programmers have to kill actors manually by developing bespoke resource management protocols. These protocols often harbor DC bugs!*

Programmers could delete their resource management code, exterminating all the DC bugs lurking inside, if only they had *distributed actor garbage collection (actor GC)* to do the job automatically.

1.1 Actor GC is Easier Than We Thought

Since the 1980s, distributed garbage collection has gained a reputation for being hard to scale and hard to understand. Both problems come down to coordination: servers essentially need to compute a "distributed snapshot" of the cluster, coordinating with one another to make sure the snapshot is "consistent" (or at least approximately consistent). Without consistency, an object might look like garbage when it isn't in reality. For decades, researchers thought actor GC was even harder than traditional GC, because each actor has its own logical thread of control.

In **Plyukhin** and Agha [2020], I proved that *actor GC is not only easier than traditional GC—there are highly parallel algorithms to solve it.* Formally, I proved that the property of being garbage in the actor model is "strongly stable" [Schiper and Sandoz, 1994]: if an actor looks like garbage, then it really is garbage—even if the garbage collector's view of the world is inconsistent! These findings showed, for the first time, that actor GC could simultaneously be simple (with formal proofs of both safety and liveness), general (without strong assumptions like causal delivery or locking mechanisms), and scalable (without requiring coordination to achieve a priori consistency).

1.2 Real Distributed Systems Have Faults

Distributed garbage collectors often assume that crashes and dropped messages can be masked, and that objects are kept in durable storage that persists across restarts. These assumptions won't hold in any of the four most popular actor frameworks without a significant performance cost. This disqualified every actor GC from the previous 30 years from even being considered for real-world use. It also left programmers to fend for themselves in situations like this one:

> *Suppose a worker on server A sends messages to servers B and C, but B does not respond. Has B crashed? Is it safe to kill the worker? What about the actors on node C?*

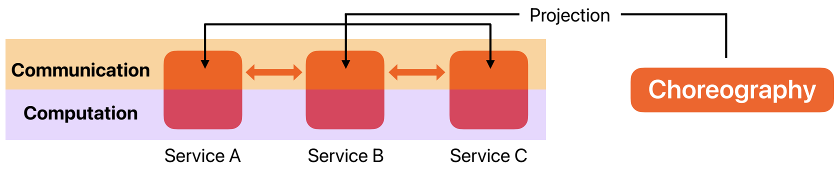
In **Plyukhin** et al. [2025a], I showed how actor GC can recover from faults without expensive synchronization mechanisms—and I even proved it correct. The approach builds on the work mentioned previously, with two key insights. First, I noticed that Akka and its open source fork, Apache Pekko, both use cluster management protocols to "exile" unresponsive nodes from the cluster. Second, I saw that messages between nodes always pass through designated ingress and egress points in each node. By collecting information at ingress and egress points, healthy nodes can reconstruct enough information about exiled nodes to collect the garbage left in their wake. The impact of these results is a simple interface for distributed resource management: whenever crashed nodes cause a subsystem to fail, the programmer can restart the subsystem elsewhere and trust that the old system's resources will promptly be reclaimed.

1.3 Paving the Way to Production-Grade Actor GC

At the end of my Ph.D. I developed **UIGC**: the first actor GC framework for Apache Pekko. Whereas past actor GCs have been tightly coupled to specific runtimes and languages, UIGC supports many different "engines" so their performance can now be compared empirically for the first time. UIGC's default engine, the *Conflict-Replicated Garbage Collector (CRGC)*, is an efficient distributed cyclic actor GC capable of recovering from faults, with safety and liveness properties that have been modeled in TLA$^+$ and proven correct. The initial performance results published in **Plyukhin** et al. [2025a] confirm that CRGC is practical, with overhead comparable to lightweight collectors like weighted reference counting. We'll revisit this topic in Section 3; for now, let's move on to choreographic programming.

2 FASTER, SAFER, AND SIMPLER WITH CHOREOGRAPHIC PROGRAMMING

Choreographic programming (CP) is an innovative solution to a fundamental problem. In distributed applications, machines have a "computation layer" that handles business logic, and a "communication layer" that interfaces with other machines. Whereas the computation layer can be modular and relatively easy to debug, the communication layer is often tightly coupled across machines, and vulnerable to complicated DC bugs. The key idea behind CP is to write a single program, called a *choreography*, that a compiler mechanically *projects* to generate the communication layer:



Since its introduction in 2012, CP has become a hot topic in the programming languages community: CP has been implemented as a library (in Haskell, Rust, Elixir, Clojure, and more), as an intermediate representation for distributed cryptography (the Viaduct language), and as a production-ready compiler that extends Java (the Choral language). Just last year, PLDI hosted the first international workshop in choreographic programming (CP@PLDI 2024). The next workshop is being co-organized by myself and my peers Andrew Hirsch and Eva Graversen.

I see choreographies as an example of *lightweight formal verification*. With CP, programmers continue to write code in a familiar language, but they also get strong properties "for free" like deadlock-freedom, deterministic parallelism, and well-typed messages. Usually in systems research, increased safety comes at the cost of performance. The impact of my research has been to show just the opposite:

> *Choreographies aren't just simpler and safer than traditional programs; my work shows they can be **faster** too!*

2.1 As Fast as Actors, but Safer

By default, CP provides "deterministic implicit parallelism": any two expressions e_1 and e_2 will be evaluated concurrently unless one expression somehow "depends" on the other. This meant choreographies couldn't implement, say, an AI pipeline that handles user requests in whatever order they arrive. From my experience with actor systems, I saw that *adding concurrency to choreographies would introduce subtle data corruption bugs in the compiler*. In our AI example, this could mean one user's embarrassing chat data gets revealed to another user, like a recent incident in ChatGPT.

In **Plyukhin** et al. [2024], I presented the necessary and sufficient conditions for safely adding concurrency to choreographies. The approach allowed me to extend Choral's standard library with support for the popular `future` abstraction. My findings are already influencing the design of new CP languages, like the Chorex DSL for Elixir, because they're an essential precursor for matching the performance of low-level abstractions like actors.

2.2 As Simple as RPCs, but Faster

Programmers are always balancing performance with maintainability. In distributed systems, even though it's more efficient to use point-to-point communication and avoid making network hops, it's often more practical to use *orchestrators*: centralized services that handle requests by making remote procedure calls (RPCs) to other services. In principle, CP languages can remedy the problem by combining the speed of point-to-point communication with the simplicity of orchestration. But this idea was impossible to use in reality because of practical limitations, like the cost of rewriting an entire application in a choreographic language.

To accelerate slow orchestrators in service-oriented architectures, our group developed the **Accompanist** [Kløvedal et al., 2026] framework for Choral. The project is driven by a talented Ph.D. student whom I proudly co-advise, Viktor Strate Kløvedal. Our key insight was to replace slow orchestrators by deploying projected Choral code into "sidecars" running alongside RPC servers; in effect the framework "decentralizes the orchestrator", replacing expensive RPCs from the orchestrator with cheap RPCs from the sidecar. Our preliminary results show a 32–55% reduction in end-to-end latency on Amazon's EKS, and a 5.9–6.7× speedup compared to Temporal, a production-grade saga orchestrator.

2.3 Toward an Optimizing Compiler for Distributed Systems

There's nothing more practical than a good theory. In choreographic programming, we use formal models to design new features and to prove those features won't cause bugs. Formal models are also the basis for optimizing compilers of so-called "functional" languages, like Haskell and OCaml, where many optimizations are just combinations of simple program rewriting steps. As I explain in Section 3, optimizing compilers for *distributed* programs could have a significant impact. So how do we design an optimizing compiler for functional choreographic languages? The first step is knowing which program rewriting rules are permitted by the formal model—but finding them is no easy task.

Prior to my work, formal models for functional choreographic languages had dozens of complex rules and still failed to predict the behavior of real-world languages like Choral. My key insight in **Plyukhin** et al. [2025b] was that, although compiled choreographic code has "strict" semantics, we expect the choreographies themselves to have a "non-strict" semantics! Using techniques developed for non-strict languages like Haskell, my collaborators and I developed a core model for choreographies with just ten simple rules. The new model has already proven to have good predictive power: we used it to discover a common structure unifying five seemingly ad-hoc rules in previous models, and from that structure we discovered three *missing* rules in past work, without which a compiler can miss optimization opportunities. The evidence strongly suggests that we found the “correct” model for optimizing CP compilers.

3 FUTURE WORK

My research makes it *easier* to write *fast*, *correct* distributed applications. With my future students and colleagues, I plan to develop new foundations for distributed programming and to put tools in the hands of developers.

3.1 Choreographic Programming With Typestate

I’ve already described DC bugs at a high level—but what do they look like in practice? Do they have any special structure that programming language researchers can use as leverage? In fact, they do.

I manually inspected a database of DC bug reports [Leesatapornwongsa et al., 2016] and found that every framework in the study used *state machines* as specifications [c.f. Hadoop, Zookeeper, HBase, and Cassandra]. I then sampled ten bugs in the database, and found a recurring pattern: seven out of ten bugs were *state machine violations*—bugs where a state machine receives an event that cannot be handled in the current state. State machine violations underlie many kinds of programming errors: sending a message before the recipient is initialized [MR-2953, MR-3274]; terminating a process too early [MR-3006, MR-4099]; violating atomicity [MR-2995, MR-3596]; and failing to account for reboots [MR-3186]. (The remaining bugs in my sample were race conditions, but not state machine violations [MR-4252, MR-4425, MR-4437].) These findings show that a significant number of real-world DC bugs could be prevented automatically with a *distributed typestate verification tool.* However, to check distributed typestate properties in traditional programming paradigms, one needs complex program annotations to reason about global aliasing. Choreographic programming can solve this problem, transforming global aliases to local aliases that can be resolved with existing flow-based techniques.

3.2 Actor GC in Production

So far, my work on actor GC has focused on how the actor abstraction is implemented. Now our attention needs to turn toward the legacy code that *uses* actors and tacitly assumes those actors need to be killed manually. Some of this old code conflicts with the garbage collector. For example, when programmers handle exceptions by restarting actors, GC can no longer determine when the restarted actor’s children are garbage. Hence the “restarting” idiom needs to be refactored by spawning new actors that replace the old ones. What other idioms that have similar problems? How hard is it to port an application to use actor GC in practice? Can automated tools do it for us? And how do we add GC to frameworks that don’t already use cluster management protocols, like Erlang, or systems that are only “inspired” by the actor model, like Ray and Apache YARN? My team will put distributed GC in every major actor framework, preventing an entire class of bugs in distributed systems used by **billions** of people.

3.3 The LLVM of Distributed Systems

Optimizing compilers are enormously impactful: they let developers write simpler programs, while users continue to benefit from performance close to expertly-written assembly code. With choreographic programming, we can extend those benefits to concurrent and distributed systems. As a simple example, imagine Service A sends a compile-time constant to Service B. An optimizing compiler could remove the message and inline the constant at Service B; but in traditional programming models, this optimization is infeasible because it requires a global analysis that spans across codebases. In choreographies, inlining the message is a trivial “peephole” optimization.

My students at SDU have already broken ground on this project. Last year, my M.Sc. advisee Christian Nesting developed an extension for the Choral compiler that infers efficient communication patterns using a constraint solver [**Plyukhin** et al., 2026]. This year, my M.Sc. advisee Malthe Petersen is redesigning the Choral compiler for better IDE integration. Could we also use runtime information to push performance even further, creating something like a choreographic JIT compiler? Can we expose Choral’s optimizations as a compilation target for higher-level languages, like a distributed analogue of the LLVM toolchain? The confluence of distributed systems and compilers research raises many interesting questions. I look forward to leading the team that finds the answers.

REFERENCES

Viktor Strate Kløvedal, **Dan Plyukhin**, Marco Peressotti, and Fabrizio Montesi. 2026. Accompanist: Simple Decentralized Service Coordination With Choreographic Programming. *Under review at PLDI* (2026).

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS '16*. https://doi.org/10.1145/2872362.2872374

André Schiper and Alain Sandoz. 1994. Strong Stable Properties in Distributed Systems. *Distributed Computing* 8, 2 (1994), 93–103. https://doi.org/10.1007/BF02280831

Dan Plyukhin and Gul Agha. 2020. Scalable Termination Detection for Distributed Actor Systems. In *31st International Conference on Concurrency Theory, CONCUR 2020*. https://doi.org/10.4230/LIPIcs.CONCUR.2020.11

Dan Plyukhin, Gul Agha, and Fabrizio Montesi. 2025a. CRGC: Fault-Recovering Actor Garbage Collection in Pekko. *Proc. ACM Program. Lang.* 9, PLDI (2025). https://doi.org/10.1145/3729288

Dan Plyukhin, Christian G. H. Nesting, Jacopo Mauro, and Fabrizio Montesi. 2026. Syncho: Synthesizing Choreographies From Orchestrations With Choreographic Programming. *In preparation* (2026).

Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. 2024. Ozone: Fully Out-of-Order Choreographies. In *38th European Conference on Object-Oriented Programming, ECOOP 2024*. https://doi.org/10.4230/LIPICS.ECOOP.2024.31

Dan Plyukhin, Xueying Qin, and Fabrizio Montesi. 2025b. Relax! The Semilenient Core of Choreographic Programming (Functional Pearl). *Proc. ACM Program. Lang.* 9, ICFP (2025). https://doi.org/10.1145/3747538

Teaching Statement

DAN PLYUKHIN

In computer science, you don't have to take anybody's word for anything. Whereas observing the structure of DNA requires X-ray diffraction, the structure of a Linux thread can be seen in `"linux/sched.h"`. Even a vague claim—like the idea that arrays are more efficient than linked lists—can be translated into a precise research question, tested on any laptop in a matter of minutes, and even formally verified with an interactive proof assistant like Lean. This radically empirical property of computer science is the foundation of my teaching. ***I teach CS students to act like scientists***: questioning received wisdom, testing their hypotheses, applying theoretical models to solve complex problems, and taking things apart to see how they work.

I've taught and advised students for almost ten years. I was a teaching assistant for thirteen semesters across five courses at the University of Illinois (UIUC) and the University of Toronto (U of T), and instructor of record for one semester at Illinois Tech (IIT). At UIUC I was nominated three times for an Outstanding TA Award, and at Illinois Tech my end-of-semester instructor rating was 4.4/5. I've also co-supervised four M.Sc. students at the University of Southern Denmark (SDU), mentored two undergraduates at UIUC, and I'm currently co-advising a Ph.D. student at SDU. Through these experiences, I developed my *empirical* and *student-centered* approach to pedagogy. Instead of "programming" my students, I try to be more like a safari guide: creating maps and giving advice to help students find their own way through the ever-evolving technological landscape.

1 LECTURE

1.1 Lesson Plan: Practice Motivates Theory and Theory Informs Practice

My day-to-day teaching motto is "there's nothing more practical than a good theory." I apply this philosophy with an *inquiry-based learning* approach. For example, one of the skills we teach in a compilers class is how to remove left recursion in LL grammars. When I was a student, the idea was presented to me "theory first", by formally defining LL grammars and then explaining why left recursion is a problem. Although I learned the material, I can't say I understood how to apply it. So years later when I taught the same topic at Illinois Tech, I started the lecture instead by live coding a recursive descent parser with the students. Together, we "discovered" why left recursion is a problem (it creates infinite loops in the parser) and we brainstormed solutions. After that introduction, the theory of LL parsing emerged naturally as a convenient way to talk about this concrete problem.

This teaching style is not just a framing device for lectures. It prepares students for how we solve problems in the real world: when a problem is too complex to solve by intuition alone, we have to step back and think at a higher level of abstraction.

1.2 Lecture: Let the Students Drive

My student-driven lecture style evolved from redesigning *CS 440: Programming Languages and Translators*, a core class at Illinois Tech. Since I lived too far from campus to teach in-person, the course had to be fully online. This was a challenge because, in my experience, online teaching is much harder to do well than face-to-face. As a TA during the 2020 lockdown, I watched attendance rates plummet across multiple courses because students would prefer to watch the recorded lectures on their own time. Low attendance was a disadvantage to the students, who missed out on direct contact with their instructor, and to the professors themselves, who felt demotivated. Some instructors responded to the problem by not posting lecture recordings at all, but this only hurt accessibility and didn't address the underlying problem.

Knowing the limitations of distance learning, I made several adjustments that made my class much more engaging compared to a traditional approach. First, I used a *flipped classroom* where I presented interactive problems and had the entire class of 60 students work together to guide me through the solution. This experiment was not without its growing pains: in my midterm evaluations, a few students pointed out they had no incentive to watch pre-class materials because I tended to repeat the same ideas during lecture. But I adapted to the feedback, for instance by making pre-class materials optional. In the end, I felt confident the flipped classroom experiment was a success: students had to interact with me to drive the lecture forward, and skipping lectures was disincentivized because it meant missing the chance to ask clarifying questions while learning. I was happy to see consistently high attendance rates throughout the semester.

My second innovation was to model my lectures after Twitch streams. I learned this trick watching the great researcher Erik Meijer, who gave a talk on Zoom with very high audience participation by encouraging us to concentrate our discussion in the chat window instead of using voice. Combined with the fact that I chose Discord for all course materials and lectures, this created an environment where students were much more willing to openly give wrong answers or admit being confused than any in-person lecture I've seen.

The unifying theme here is an opinionated approach to pedagogy: I spend less time presenting *concepts*, so I can spend more time teaching *how to find solutions independently*. Teaching students a few extra UNIX commands is not as good as teaching them how to find information in the UNIX manual.

1.3 Review: Using Feedback to Improve Lectures

I'm always curious to see what concepts are giving students the most trouble. My favorite example comes from a lecture I gave at Illinois Tech about the λ-calculus. Up to that point, the students had been learning the Haskell programming language and following along nicely. But early in the λ-calculus lecture, I noticed problems: many students were reporting being "confused" during the lecture and by the pre-class reading, with some even admitting they "hate" the new syntax. I was surprised, since the λ-calculus is just a stylized subset of the Haskell language they had already learned so well! I realized there were two subtle problems with my teaching approach:

- *Unclear motivation:* Although my lecture alluded to applications of λ-calculi, I never gave the students a chance to see for themselves. This made several students demotivated.
- *Cognitive overload:* Students were already struggling with the mapping between Haskell's ASCII (`\x -> M`) notation and the mathematical $\lambda x.M$ syntax. Since I taught higher-level concepts like variable capture *on top* of the mathematical syntax, students easily became overwhelmed.

I'm thankful I took a student-centered teaching approach, because otherwise I might not have known there was a problem until the midterm. Using the extra feedback, I made sure to give extra time for the material to sink in. Teaching the material today, I would avoid the roadblocks:

- *Motivate theory with practical examples:* Equational reasoning, macros, type systems, and compiler optimization are all examples where λ-calculi arise in practice. Introducing these ideas first can help motivate the theory.
- *Improve instructional scaffolding:* Variable capture is not just a feature in the λ-calculus; programmers encounter it in the macro systems of real languages. We can improve instructional scaffolding by removing the learning dependency between variable capture and λ-calculus syntax.

1.4 Future Teaching Interests

I'm keen on teaching a wide range of classes, including but not limited to: compilers, operating systems, distributed systems, programming language theory, introductory computer science, and discrete mathematics. Aside from my teaching approach, my biggest contribution to these courses will be to create a platform of interactive assessments that can be reused across semesters and instructors. As I explain in Section 2, I've found that students benefit tremendously from having a huge playground of low-stakes games to hone their skills. For example, at UIUC we developed an online text-based assessment in which students constructed LR parsing tables. Students loved the assessment because they could experiment with the tool and receive instant feedback when they made mistakes. Instructors loved it too, because it left us with more time for high-quality one-on-one instruction.

Given the opportunity, I'm also interested in designing new courses. Many departments don't yet teach *empirical software engineering*: how to fix complex pieces of software that exhaust resources or produce incorrect results. Such a course could emphasize profilers, dynamic tracing tools, measurement bias, data-oriented design, and so on. At a graduate level, I'd like to design a class in my area of expertise: *distributed programming and concurrency theory*. The course would cover concepts like actors (in the context of Apache Pekko, Ray, or Erlang), model checking and linear temporal logic (with TLA$^+$), process calculi and bisimulation (with Lean), and session types and multiparty languages (like Choral and ScalaLoci) using a mix of textbooks, primary source articles, and student-driven inquiry.

2 ASSESSMENTS

2.1 Assessment Design

Assessments are where the rubber meets the road. I use *quizzes*—frequent low-stakes take-home assessments with unlimited retakes—so students can check their understanding of key ideas in the latest lectures (c.f. "understanding and applying" in Bloom's taxonomy). Larger *assignments* give students a chance to chain ideas together and apply them in a realistic context (c.f. "analyzing, evaluating, and creating" in Bloom's taxonomy). Lastly, *exams* are a necessary evil: we need them to evaluate the student while controlling for extra help they may be getting from LLMs or other students. After all, exams aren't just a measure of how well students are learning—they show how well the professor is teaching!

To give students the best learning experience, I try to be cautious about using exams. Exams are stressful and they unfairly penalize some students over others. Many professors and TAs don't appreciate the severity of the problem because selection bias means we tend to be good test takers. At Illinois Tech, I accommodated students with test anxiety by dividing each exam into "zones" that assessed core competencies. Every zone appeared twice—once on a midterm and again on the final—and a student's end-of-semester exam score was the sum of the maximum score for each zone across exams. As a result, students who did well in midterms could skip the final and students who wanted to boost their grade could use the final as a second attempt. Simple gestures like this can make life easier for everyone.

2.2 AI Assistants and Plagiarism

AI assistants are having a significant impact on education. In some ways, they're more of the same: students have always found ways to cheat themselves out of learning, like relying on their friends. But sometimes a difference in scale can be a difference in kind: many students are using LLMs as a "shadow TA", guaranteed not to pass judgment and instantly available at all hours of the day. These shadow TAs often give subtly wrong answers, and are not as scrupulous as real TAs about letting students think for themselves. But LLMs can also save human instructors time answering menial questions ("missing semicolon on line 42"), allowing us to focus on more conceptual misunderstandings.

I'm curious and optimistic about the creative possibilities of assessments where we *assume* students are assisted by AI. With these tools, the student takes on a more "managerial" role where they spend less time typing and looking through dense API documentation, and more time thinking about architectural design patterns and doing code review. To paraphrase Dijkstra, I'm optimistic that AI assistants can promote students from *telescope scientists* into real *astronomers*.

3 ADVISING AND MENTORSHIP

It's essential for advisees to have a tangible sense of what problem they're trying to solve at the moment, what the success criteria look like, and a short feedback loop to confirm progress or report concrete setbacks. The types of problems they address also need to change over time; at the beginning of a recent M.Sc. project, I gave prescriptive and concrete goals that were sometimes as simple as "get the code to compile" or "reproduce this bug". The joy of advising came from later stages, when that student was confident and experienced enough to tackle open-ended problems and began presenting unexpected solutions. It's tempting to micromanage advisees and make sure their work meets our expectations—but because I gave the student enough room to work on his own, he reported feeling a real sense of ownership in his work.

Aside from the ability to do great research, the other skill I want my advisees to take away is *how to be a great communicator*—both in writing and presentations. This involves control over grammar and syntax, the ability to break down and organize information in a logical order, the ability to imagine many different points of view, the ability describe an idea visually, and so on. Great communication can't be learned overnight, so I encourage my advisees to write early and often so I can give frequent feedback. I also ask advisees to give presentations frequently, for the same reason; although the aforementioned M.Sc. student was initially reluctant to present, he was glad to receive the early feedback. I very much look forward to helping more students along their scientific journey.

University of Southern Denmark
Department of Mathematics and Computer Science
Odense, Denmark

dplyukhin.github.io
dplyukhin@imada.sdu.dk
@dplyukhin.bsky.social

Dan Plyukhin

Citizenship: USA, Canada

Key Interests

Research: Programming Languages, Distributed Systems, Choreographic Programming, Actors
Teaching: Compilers, Operating Systems, Distributed Systems, Discrete Math, Algorithms

Education

Ph.D. Computer Science at University of Illinois Urbana-Champaign (UIUC) · · · · · · · · · · · · · · 2024
Thesis: Fault-Tolerant and Fault-Recovering Garbage Collection for the Actor Model
Committee: Gul Agha (chair), Indranil Gupta, Tianyin Xu, and Philip Haller

Ba.Sc. Computer Science & Mathematics at University of Toronto (U of T) · · · · · · · · · · · · · · · · 2017
Graduated honours with high distinction

Academic Appointments

Postdoctoral Researcher at University of Southern Denmark (SDU) · · · · · · · · · · · · · · 2022–Present
Supervisor: Fabrizio Montesi

Adjunct Professor at Illinois Institute of Technology (Illinois Tech) · · · · · · · · · · · · · · · · · · Fall 2021
Instructor of record for CS 440: Programming Languages and Translators (60 students, 1 TA)

Major Software Projects

UIGC distributed actor GC for Apache Pekko · · · · · · · · · · · · · · · github.com/dplyukhin/uigc-pekko
Principal author and maintainer

Choral compiler for choreographic programming with Java · · · · · github.com/choral-lang/choral
Contributor, maintainer, and author of the *Ozone* concurrency library

Publications

Journal Articles

CRGC: Fault-Recovering Actor Garbage Collection in Pekko.
Dan Plyukhin, Gul Agha, and Fabrizio Montesi. *Proceedings of the ACM on Programming Languages* PLDI (2025). doi.org/10.1145/3729288.

Relax! The Semilenient Core of Choreographic Programming (Functional Pearl).
Dan Plyukhin, Xueying Qin, and Fabrizio Montesi. *Proceedings of the ACM on Programming Languages* ICFP (2025). doi.org/10.1145/3747538.

A Scalable Algorithm for Decentralized Actor Termination Detection.
Dan Plyukhin and Gul Agha. *Logical Methods in Computer Science* 1 (2022). doi.org/10.46298/lmcs-18(1:39)2022.

Peer-Reviewed Conference Papers

Ozone: Fully Out-of-Order Choreographies.
Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. *38th European Conference on Object-Oriented Programming, ECOOP 2024*. doi.org/10.4230/LIPIcs.ECOOP.2024.31.

Scalable Termination Detection for Distributed Actor Systems.
Dan Plyukhin and Gul Agha. *31st International Conference on Concurrency Theory, CONCUR 2020*. doi.org/10.4230/LIPIcs.CONCUR.2020.11.

Peer-Reviewed Workshop Papers and Extended Abstracts

Poroutines: The Essence of Choreographic Programming?
Dan Plyukhin. *1st International Workshop on Choreographic Programming, CP@PLDI 2024*. Extended abstract.

Concurrent Garbage Collection in the Actor Model.
Dan Plyukhin and Gul Agha. *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018*. doi.org/10.1145/3281366.3281368.

Other Manuscripts

Accompanist: Simple Decentralized Service Coordination With Choreographic Programming.
Viktor Strate Kløvedal, **Dan Plyukhin**, Marco Peressotti, and Fabrizio Montesi. (2025). Under review at PLDI 2026.

Fault-Tolerant and Fault-Recovering Garbage Collection for the Actor Model.
Dan Plyukhin. PhD thesis. University of Illinois Urbana-Champaign, USA (2024). hdl.handle.net/2142/124328.

Talks

(*Excluding talks for the above papers*)

Web Architecture Through The Ages: From Servers to Microservices 2025
Invited talk at *University of Regensburg* (Software Engineering I)

Garbage Collection in Erlang vs JVM/Akka 2023
Podcast guest on *Elixir Wizards* by SmartLogic

Actors, GADTs, and Burnout 2023
Podcast guest on *Type Theory Forall*

A Language for Low-Latency Distributed Systems 2023
Invited talk at *Purdue PurPL Seminar*

Making It Easier to Implement Correct Distributed Algorithms 2022
Student talk at *BehAPI Summer School*

Capabilities for Flexible and Concurrent Garbage Collection of Actors 2018
Contributed talk at *Midwest PL Summit 2018*

An Introduction to Network Programming 2018
Workshop at *Illinois CS SAIL* (high school outreach event)

Teaching

Instructor

CS 440: Programming Languages and Translators (Illinois Tech) . . . . Fall 2021
Sole instructor of record with one TA. Taught 60 students, fully remote using Discord and PrairieLearn. Developed curriculum, assignments, and lectures. Teaching reviews available at: dplyukhin.github.io/files/plyukhin-cs440-fa2021-evals.pdf

Midterm instructor rating: **4.52/5** (29 reviews, 48% response)
Final instructor rating: **4.40/5** (10 reviews, 17% response)

Teaching Assistant

CS 421: Programming Languages and Compilers (UIUC) · · · · · · · · · · 2017–2022
Taught 9 semesters, both remote and in-person. Held office hours and contributed to course development on Coursera and PrairieLearn platforms.

CS 425: Distributed Systems (UIUC) · · · · · · · · · · Fall 2019
Online M.Sc. section, Coursera. Graded exams and assignments, held office hours.

CSC 240: Enriched Theory of Computation (U of T) · · · · · · · · · · Spring 2017
Taught a biweekly lab, graded assignments and exams, held office hours.

CSC 324: Principles of Programming Languages (U of T) · · · · · · · · · · Fall 2016
Taught a weekly lab, graded assignments and exams, held office hours.

MAT 246: Abstract Mathematics (U of T) · · · · · · · · · · Spring 2016
Taught a weekly lab, graded assignments and exams, held office hours.

Supervision and Mentorship

Doctoral

Viktor Strate Kløvedal† (SDU) · · · · · · · · · · 2024–Present

Masters

Malthe Hedelund Petersen⋆ (SDU) · · · · · · · · · · 2025–2026

Christian Hviid Nesting⋆ (SDU) · · · · · · · · · · 2024–2025

Steven Kolbeck Ensted† (SDU) · · · · · · · · · · 2023–2024

Jonas Bruun Plesner† (SDU) · · · · · · · · · · 2023–2024

Mathias Jensen† (SDU) · · · · · · · · · · 2023–2024

Undergraduate

Charles Kuch⋆ (UIUC) · · · · · · · · · · 2020

Jerry Wu⋆ (UIUC) · · · · · · · · · · 2020

⋆: primary supervisor †: co-supervisor

Honors and Awards

Outstanding Teaching Assistant Award (UIUC) (*Nomination*) · · · · · · · · · · 2022

Outstanding Teaching Assistant Award (UIUC) (*Nomination*) · · · · · · · · · · 2021

Outstanding Teaching Assistant Award (UIUC) (*Nomination*) · · · · · · · · · · 2019

Dean's List (U of T) · · · · · · · · · · 2017

Undergraduate Research Experience (U of T) (*Research Grant*) · · · · · · · · · · 2017

University College Special In-Course Scholarship (U of T) · · · · · · · · · · 2017

Dean's List (U of T) · · · · · · · · · · 2016

Dean's List (U of T) · · · · · · · · · · 2014

Service

Institutional

Co-Chair for the *2nd Workshop on Choreographic Programming* at PLDI 2026 · · · · · · · · · · · 2026

Program Committee Member for ICST (Poster Track) · 2025

External Reviewer for POPL 2026, EXPRESS/SOS 2025, JLAMP 2025, Journal of Computer Languages 2023, ISPDC 2022, JLAMP 2019, COORDINATION 2018 ·

Community

Co-Host of the *Type Theory Forall* podcast · 2023–Present

Seminar Organizer (SDU) for the *ACP Section* · 2024–Present

Member (UIUC) of the *Education Justice Project (EJP)* · 2021–2024

Mentor (UIUC) for the *Undergraduate Research Apprenticeship Program* · · · · · · · · · · · 2019–2020

Instructor (UIUC) for the *CS SAIL* high school outreach event · 2018

Seminar Organizer (U of T) for the *Undergraduate Theory Group (UTG)* · · · · · · · · · · · · 2015–2017

References

Gul Agha (UIUC) Research Professor and Professor Emeritus · · · · · · · · · · · · · agha@illinois.edu

Fabrizio Montesi (SDU) Full Professor · fmontesi@imada.sdu.dk

Mattox Beckman (UIUC) Teaching Associate Professor · · · · · · · · · · · · · · · · · mattox@illinois.edu

Tianyin Xu (UIUC) Associate Professor · tyxu@illinois.edu