Jonathan Mace - Research Statement

November 2021

My research is in the areas of distributed systems, networking, and operating systems. I am particularly focused on *end-to-end reliability and robustness* of modern Internet-scale applications. These applications are complex global-scale distributed systems that comprise many different layers of inter-operating components. Today even the largest, best-equipped Cloud service providers struggle to keep their systems running reliably, and continually suffer from unanticipated and costly service outages. The ultimate goal of my research agenda is to inform a new generation of distributed systems that attain end-to-end reliability and robustness *by design*.

My research tackles this challenge along three axes. First, I develop tools for *troubleshooting distributed systems at runtime*, that enable system operators to detect, investigate, and react to cross-component problems when they occur in a running system. Second, I investigate techniques for *automatically mitigating cross-component performance problems*, with mechanisms that dynamically orchestrate resource and scheduling decisions end-to-end across a system at runtime. Third, I develop infrastructure for *empirically discovering cross-component design problems prior to deployment*, enabling developers to rapidly build, reconfigure, and validate full-system designs. Taken together, these efforts represent three complementary pillars of research for realizing my vision of end-to-end reliable and robust distributed systems:

(P1) End-to-end Introspection. What system support is needed for observing and troubleshooting cross-component problems in production systems?

(P2) End-to-end Performance Management. How should components of a distributed system inter-operate to to ensure reliable end-to-end performance?

(P3) End-to-end Design Validation. Can we discover preventable cross-component design flaws in systems prior to their deployment?

Approach. A common theme shapes my research: reliability in distributed systems is inherently a *cross-component* concern. The end-to-end health of a system is influenced by all of the system's constituent components; symptoms of problems are often far-removed from root causes; and decisions by one component can have cascading effects in other components. Yet by contrast, modern applications are designed to restrict cross-component visibility and interactions, instead emphasizing loosely-coupled components, narrow APIs, and a strong separation of concerns. This dichotomy surfaces repeatedly in my research: troubleshooting tools must establish cross-component visibility; resource and scheduling mechanisms must communicate and co-operate across component boundaries; and empirical testing infrastructure must reproduce cross-component behavior to elicit cross-component phenomena and problems.

In what follows I will present an overview of my research work accomplished so far and my plan for ongoing and future research, structured according to the three pillars outlined above.

(P1) End-to-end Introspection

Deployed distributed systems will inevitably encounter unanticipated performance and correctness problems, which if left unaddressed can lead to system outages and significant loss in revenue. The first direction of my research improves the reliability and robustness of distributed systems with tools for *end-to-end introspection*. These tools enable system operators to identify and react to problems as quickly as possible

when they arise; they give system operators visibility to introspect running systems and anticipate the potential for problems to arise; and they enable system developers to explore hypotheses about potential optimizations or redesigns to prevent problems from occurring. My research has developed the core runtime components needed for efficiently tracing systems, both for troubleshooting the behavior of systems in aggregate, and for troubleshooting rare outlier and edge-case behaviors.

Causal Metadata Propagation. Troubleshooting distributed systems is notoriously difficult because problems *cascade* between components. For example, a slow or overloaded backend service can cascade into latency, queueing, and load imbalance effects in others, for both direct and indirect dependencies. Traditional troubleshooting tools for single-machine systems are ineffective in a distributed setting because their visibility doesn't extend beyond the confines of a single machine. As a result it is profoundly difficult to pinpoint the root causes of problems when they span multiple components. Addressing this limitation, causal tracing tools record the events occurring in different components of a distributed system, and use a technique called *causal metadata propagation* for capturing the relationships and ordering of causally-related events *across* component boundaries.

My research has developed the foundational techniques for establishing cross-component causality in distributed systems. In my dissertation work I proposed an abstraction called *Baggage*, now widely used in practice, which describes how information should be opaquely passed between components of a distributed system. Baggage is a general-purpose abstraction that can be used by any cross-component tools deployed in a distributed system, including the troubleshooting tools described later in this section. I initially proposed Baggage as a component of the Pivot Tracing troubleshooting tool (described below) [SOSP15], and later refined the ideas into an abstraction layering called the Tracing Plane [EUROSYS18]. The work as a whole is summarized in my PhD thesis [PHD18], and it was recognized with the Honorable Mention for the 2018 SIGOPS Dennis Ritchie Doctoral Dissertation Award.

This research has attracted significant interest from the community and industry, and following my PhD studies I have sought to maximize the practical impact of this work. Baggage is now a de facto component of causal tracing tools. Baggage is present in all major open-source implementations such as OpenTracing and Jaeger, and it has enabled the development of novel cross-component troubleshooting tools both in industry and in subsequent research. I co-authored an early survey on distributed tracing [SOCC16] and since then I have been actively involved in developing today's open-source standards for tracing infrastructure as part of the OpenTracing advisory board. I recently distilled much of this knowledge by co-authoring *Distributed Tracing in Practice*, the industry standard textbook on the topic which summarizes the state of the art in causal tracing and lessons learned over the course of developing the OpenTracing Standard [BOOK20].

Troubleshooting Aggregate System Behavior. Leveraging causal metadata propagation techniques, my research has developed troubleshooting tools for several broad classes of problems. My initial work focused on tools for understanding the behavior of a system *in aggregate*. One such tool is Pivot Tracing [SOSP15], which enables a system operator to interactively troubleshoot a live distributed system by issuing SQL-like queries about system state. In Pivot Tracing I observed that the information needed to troubleshoot cross-component problems is often relatively minimal, but inaccessible due to a lack of cross-component visibility. For example, an operator diagnosing a resource issue might wish to measure the resource consumption of a backend component while grouping and filtering by frontend API identifiers, so as to attribute the bottleneck to a culprit workload; however backend components are oblivious to frontend identifiers and frontends lack access to backend resource metrics. Pivot Tracing overcomes this limitation by combining causal metadata propagation and dynamic instrumentation. Using Pivot Tracing, a system operator can use a simple SQL-like interface to define and measure arbitrary system metrics, and group, filter, and aggregate those metrics according to arbitrary identifiers across all system components. I presented Pivot Tracing at SOSP 2015 [SOSP15] where it received the Best Paper Award. Pivot Tracing attracted substantial attention, with versions of the SOSP paper and talk invited to appear at USENIX ATC, the USENIX ;login: magazine, ACM Transactions on Computer Systems [TOCS18], and the Research Highlights of the Communications of the ACM [CACM20].

I subsequently expanded these ideas in my work on Canopy [SOSP17], a causal tracing system deployed

in production at Facebook. The goal of Canopy was to support aggregate querying of system data similar to Pivot Tracing, but to do so atop verbose end-to-end trace data. In this way, system operators can troubleshoot the symptoms of problems in aggregate, while still being able to drill down into a detailed execution history upon identifying a problem. Like Pivot Tracing, Canopy's design overcomes a delicate trade-off of capturing the data useful for the troubleshooting task, while minimizing the performance impact on the running system. Canopy was presented at SOSP 2017 [SOSP17] and is currently in use throughout Facebook's infrastructure, where it captures and processes over a billion traces per day. Since its publication, Canopy's design for extracting and aggregating metrics from traces has directly informed the backend design of today's open-source causal tracing tools.

Edge-Case Troubleshooting. Over the course of this research, it became clear that aggregate troubleshooting was only one side of the coin. Often, the most pernicious problems in distributed systems are ones that are triggered extremely infrequently. In my subsequent research I have been developing troubleshooting tools centered on understanding *outlier* system behavior. Despite being rare, troubleshooting outlier behavior is extremely important because unexplored code paths or untested conditions have the potential to wreak havoc and bring down entire systems.

The crux of outliers is detecting them. Ahead of time, it is impossible to determine that a specific request will trigger a rare edge-case behavior. After the fact, it is impossible to go back and collect detailed data about a specific request if we were not already recording it. Existing designs for aggregate analysis often miss edge-cases entirely, because they intentionally trace only a very small representative sample of requests, a decision made ahead of time as a trade-off for computational efficiency. To overcome this fundamental barrier, my recent research has redesigned and generalized the underlying tracing mechanisms, to produce the first tracing tools to comprehensively support edge-case troubleshooting.

My initial research on edge-case troubleshooting looked at how to detect rare and outlier requests, and I devised unsupervised approaches for distinguishing edge-case executions from common-cases, with approaches based on graph kernels [SOCC18] and graph embeddings [SOCC19]. These techniques enabled new troubleshooting tools that automatically guide operators towards edge-case exemplar traces, and substantially reduced their backend retention costs by automatically discarding large amounts of redundant common-case trace data.

A simple yet important observation emerged from this work: for tracing edge cases, we only need to detect symptoms, not root causes. Although root causes are many, varied, and difficult to predict, they only manifest as a small set of well-understood symptoms such as high latency, timeouts, and exceptions. My recent work on Hindsight [UR2] exploits this idea with a new tracing design called *retroactive tracing*. Hindsight traces edge-cases by detecting the *symptoms* of outliers, rather than trying to identify root causes. Symptoms can be quickly and programmatically detected soon after a problem occurs. Ahead of time, Hindsight exhaustively traces all executions in the system, implicitly tracing all edge-cases requests. By default, however, Hindsight cannot afford the computational cost of centralizing and persisting all of these traces, so beyond generating data into local memory, Hindsight takes no further action and eventually overwrites old data with new. This presents a window of opportunity: if we can programmatically detect the symptoms of a problem within this short window of time, Hindsight can go back and perform a distributed collection of the relevant slice of trace data before it disappears. This approach is profoundly effective, capturing close to 100% of rare and outlier behaviors in experiments, compared to only a fraction of a percent for conventional tracing systems. Hindsight embodies a rethink of the internal structure of telemetry systems for distributed systems, tailored towards only capturing data that will be useful for troubleshooting. Hindsight is currently under review [UR2].

User-Centric Troubleshooting. The main contributions of my research to date have been the technical mechanisms necessary for cross-component troubleshooting. As this technical foundation matures, my focus for future work is shifting towards effectively *using* the data that is captured. To this end I recently conducted a comprehensive design study with practitioners across two large internet companies to characterize current troubleshooting practices [UR1], and insights from the study are informing a more user-centric perspective in my ongoing research.

(P2) End-to-end Performance Management

The second pillar of my research focuses on runtime techniques to actively mitigate end-to-end performance problems. Performance problems are a major threat to end-to-end reliability and robustness, and many large-scale system outages have been the result of degraded performance and internal system bottlenecks, rather than outright crashes or correctness issues. The crux of performance in distributed systems is that desirable performance properties are *global*, such as fair sharing of resources, end-to-end latency goals for top-level requests, and performance isolation between tenants, yet it is *local* decisions by the constituent system components that cumulatively determine the fate of executions. The whole is often less than the sum of parts: decisions that are locally sensible (e.g. for request scheduling, admission control, timeouts) may be globally counter-productive, and lead to unacceptable phenomena such as high tail latency and low resource utilization. My research tackles this problem through the design of cross-component scheduling and orchestration mechanisms, collectively *end-to-end performance management frameworks*, to ultimately establish reliable and robust end-to-end performance.

Reactive Policies. In my initial work I designed a system called Retro [NSDI15], for coordinating resource policies end-to-end across a distributed system. The key idea of Retro is to extract resource management into a separate, dedicated layer, to outsource scheduling decisions to a logically centralized controller that has full view of the end-to-end system. To achieve this, Retro continually measures resources consumed across all system components, reports this information to the central controller in realtime, and reactively co-ordinates rate-limiters embedded across the system. Retro introduces a narrow set of abstractions for resources and rate-limiting so as to broadly apply to a wide range of system designs. The prevailing contribution of this work is Retro's design for centralizing cross-component performance management policies. Retro is intentionally general-purpose and based on simple reactive control loops: observe resource consumption, consult the global performance policy, and adjust rate limits in response. Retro demonstrates that simple rate-limiting is effective for enforcing resource policies at a coarse granularity, such as bottleneck fairness, latency guarantees, and dominant resource fairness.

Consolidating Choice. The next challenge was to mitigate short-term effects like rapid workload fluctuations, head-of-line blocking, and cascading admission backlogs, which arise over short time scales and inhibit the long-term convergence of reactive techniques like Retro. In follow-on work I moved beyond the simple rate-limit enforcement of Retro and proposed a scheduling algorithm called Two-Dimensional Fair Queueing [SIGCOMM16], which pro-actively stratifies requests by size and tracks admitted work to the system, ultimately reducing performance variability by avoiding head-of-line blocking and resource starvation.

However, it was my most recent work on Clockwork [OSDI20] that finally overcame the fundamental limitations of reactive best-effort enforcement. Clockwork is a distributed DNN serving system designed for end-to-end predictable performance. Instead of super-imposing reactive performance management techniques, Clockwork has proactive end-to-end performance predictability as a first-class design concern. To achieve this, the system's design must eliminate any sources of performance variability, or centralize the choices that lead to variability, such as scheduling and admission control. I called this design approach *consolidating choice*, and it led to a system design where a centralized scheduler is able to accurately estimate the consequences of any scheduling decision. Leveraging this, Clockwork proactively schedules requests with high confidence that performance goals can be met. This prevents unexpected performance fluctuations, backlogs of work, or tail-latency effects. Clockwork's results for tail-latency predictability were astounding compared to systems built with traditional best-effort reactive schedulers, with 99.999th percentile latency remaining within 10% of the median even at close to 100% resource utilization. Clockwork was published at OSDI 2020 [OSDI20] where it received the Distinguished Artifact award, and a proposal based on Clockwork was a finalist for the 2020 Facebook Faculty Award.

Beyond Human-Optimized Performance Management. Several themes have emerged from my work so far on end-to-end performance management. First is that centralized decision making is effective and

intuitively appealing for achieving end-to-end performance goals in distributed systems. Second is that global resource management policies depend highly on the enforcement mechanisms present in a system, the available observability signals such as resource consumption, and the global performance goal being enforced. However, there are also significant confounding factors absent from the traditional single-process setting: dynamic interactions between system components, continually-evolving system topologies, and the presence of multiple competing performance objectives. Combined, these factors make reliable end-to-end performance still exceedingly difficult to achieve, and it is no surprise that deployed systems today still employ broad-stroke techniques such as resource over-provisioning, in an attempt to alleviate symptoms of ineffective performance management.

I am thus driven by the belief that human-optimized scheduling techniques based on hand-crafted heuristics are a poor fit for modern distributed systems. Instead, I firmly believe that breakthroughs in reliable end-to-end performance will come from the use of *learned performance management* in place of human-optimized schedulers. I have begun to explore this direction in my ongoing work. Building off my work on Clockwork, I am currently exploring the design of Reinforcement Learning (RL) agents for end-to-end performance management; under this lens the problem becomes one of co-designing system mechanisms (action space), observability signals (state space), and codifying performance goals (rewards). I have devised an approach based on accurate system simulation that leads to capabilities beyond what can be achieved today by human-optimized schedulers. In particular, learned schedulers can enforce diverse per-tenant SLOs simultaneously, for workloads and system topologies that change dynamically over time. Long-term I aim to impact systems builders by discovering abstractions and system designs for integrating learned performance management into distributed systems.

(P3) End-to-end Design Validation

The third pillar of my research aims to change the way we discover and resolve cross-component problems in a system's design, by empirically unearthing those problems *before the system is deployed*. The first two pillars of my research work have made it increasingly apparent that many problems in deployed systems are inadvertently the result of conflicting or sub-optimal design choices that could have been identified and addressed a priori at development time. Yet today, few options exist for developers to test the end-to-end impact of design choices beyond simply deploying the system. By the time the problems manifest in a deployed system, it is usually too late to go back and re-design the system as a whole, due to enormous development burden, inertia, and the potential that old problems get replaced with new ones.

Flexible Full-System Design. I am currently developing a full-system compilation framework called Millenial [IP1]. We observe that the key to mitigating cross-component problems at development time is to make it easy to *change* aspects of a system's composition, so that the developer can experiment with the performance consequences of different design choices prior to deployment. These aspects are the system's *scaffolding*: pieces of the system that are independent and orthogonal to the flow of application-level logic but dictate the concrete end-to-end execution behavior of the system; for example replication techniques, RPC frameworks, threading libraries, load balancers, timeouts and retries, schedulers, placement and affinity, and many more. Today, developers tightly couple application code with scaffolding choices very early in the development process. By the time a cross-component problem arises, it is far too late to rip out the foundations and try again with different choices.

By contrast, Millenial makes it easy to re-design a system by *not binding application logic to infrastructure choices*. This is achieved with an internal programming abstraction that cleanly separates application-level concerns (the logic dictating the flow of an execution) from scaffolding. A system compiled using Millenial is ultimately indistinguishable from a hand-crafted system. Yet, a user of Millenial can go back and trivially *change* scaffolding choices or the composition of the system, to easily regenerate an entirely new variant of the system. Millenial can be used to quickly explore the design space of different scaffolding choices, without wasting developer effort on what ought to be simply boilerplate. Millenial itself is designed to be

modular and extensible, for easily integrating entirely new scaffolding dimensions and choices that may arise in the future.

Data-Driven Exploration. The ability to change a system's composition is an important step towards addressing cross-component problems at development time. For example, Millenial enables approaches for empirically exploring the design space of different system choices, because users can quickly recompile and redeploy variants of a system with different choices. In this direction, I am currently collaborating with researchers at Twitter on data-driven approaches to system design, where we are using production trace data to estimate the potential latency impact of changing the geographical placement of different services. Millenial gives us the ability to empirically evaluate the quality of proposed placement choices, and feed the results back to a data-driven exploration loop.

Prototyping Cross-Component Tools. The ability to easily change a system's scaffolding choices also makes it possible to prototype and integrate new cross-component tools for runtime mitigation. For example, I am currently collaborating with researchers at IST - ULisboa on cross-component consistency, a new kind of consistency violation that can occur when executions traverse multiple mutually-independent datastores. We have developed a tool called Antipode that enforces *cross-system causal consistency*, a new consistency model that we propose. Millenial enables us to prototype, deploy, and empirically demonstrate Antipode across a wide range of diverse system designs, and moreover allows existing Millenial applications to trivially integrate Antipode.

Open-Source Benchmarking. Millenial has the potential to significantly impact the research community by providing the first platform for evaluating system infrastructure for the *entire design space* of microservice applications. A conundrum for research and development of novel scaffolding is that there is no single canonical distributed application design, and any given application represents just a single point in the large design space. For researchers, a prototype that works with one set of scaffolding choices might not generalize to others. Yet the time burden of integrating a prototype and changing scaffolding choices, for a diverse range of applications, is monumental.

Millenial's flexibility eliminates this barrier. Millenial enables broad experimentation across the design space of microservice applications by enabling users to trivially toggle and configure scaffolding choices. A developer or researcher prototyping new infrastructure component need only integrate it with Millenial's compiler abstractions once, for it to then work with any existing Millenial application. So far we have ported *all* major open-source microservice benchmarks to Millenial, including the DeathStar and TrainTicket benchmarks, along the way identifying and fixing multiple cases of sub-optimal design and cross-component performance problems. Beyond existing benchmarks, we have begun development of several new, large-scale benchmarks based on industry trace data and input from industry partners. We hope that our work in this direction will pave the way to more rigorous and generally applicable infrastructure contributions from the research community.

Looking Ahead

Today it is easier than ever to build and deploy large-scale distributed systems. Yet the same cannot be said for our ability to keep those systems running reliably: we pay the price with systems that are at times inefficient, unpredictable, and unreliable. Even the largest, best-equipped Cloud service and Internet-scale application providers — with virtually infinite resources and some of the brightest minds and very best engineers on staff — struggle to keep their systems running reliably. *All* major internet companies and cloud providers suffer outages to this day.

Long-term, the goal of my research is to reverse this situation. There is great promise in addressing the reliability of distributed systems with more principled design, and in my ongoing research I strive to raise the bar for empirical end-to-end distributed system evaluation and testing to the quality established for single-node applications.

References

[NSDI15] **Jonathan Mace**, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. "Retro: Targeted Resource Management in Multi-tenant Distributed Systems". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. May 2015.

[SOSP15] **Jonathan Mace**, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2015. ***Best Paper Award***.

[SIGCOMM16] **Jonathan Mace**, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. "2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services". In: *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM)*. Aug. 2016.

[SOCC16] Raja R Sambasivan, Ilari Shafer, **Jonathan Mace**, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. "Principled Workflow-Centric Tracing of Distributed Systems". In: *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*. Oct. 2016.

[SOSP17] Jonathan Kaldor, **Jonathan Mace**, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. "Canopy: An End-to-End Performance Tracing And Analysis System". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2017.

[TOCS18] **Jonathan Mace**, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *ACM Transactions on Computer Systems (TOCS)*. Vol. 35. 4. 2018.

[EUROSYS18] **Jonathan Mace** and Rodrigo Fonseca. "Universal Context Propagation for Distributed System Instrumentation". In: *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys)*. Apr. 2018.

[PHD18] **Jonathan Mace**. "A Universal Architecture for Cross-Cutting Tools in Distributed Systems". Ph.D. Thesis. Brown University, May 2018. ***Honorable Mention - SIGOPS Dennis M. Ritchie Doctoral Dissertation Award***.

[SOCC18] Pedro Las-Casas, **Jonathan Mace**, Dorgival O. Guedes, and Rodrigo Fonseca. "Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay". In: *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC)*. Oct. 2018.

[SOCC19] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and **Jonathan Mace**. "Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering". In: *Proceedings of the 10th ACM Symposium on Cloud Computing (SoCC)*. Nov. 2019.

[CACM20] **Jonathan Mace**, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *Communications of the ACM (CACM)*. Vol. 63. 3. 2020.

[BOOK20] Austin Parker, Daniel Spoonhower, **Jonathan Mace**, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Apr. 2020.

[OSDI20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and **Jonathan Mace**. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2020. ***Distinguished Artifact Award***.

[IP1] Vaastav Anand, Gerd Alliu, Deepak Garg, Antoine Kaufmann, and **Jonathan Mace**. "Millennial: Modular Microservice Macrobenchmarks". 2021. *In Preparation*.

[UR1] Thomas Davidson, Emily Wall, and **Jonathan Mace**. "Design Guidelines for Visualisation in Distributed Tracing Tools". 2021. *Under Review*.

[UR2] Lei Zhang, Vaastav Anand, Ymir Vigfusson, and **Jonathan Mace**. "The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems". 2021. *Under Review*.