Self-Defining Systems

Thomas Anderson, Ratul Mahajan, Simon Peter, Luke Zettlemoyer

Paul G. Allen School of Computer Science & Engineering, University of Washington

December 2025

1 Introduction

The extraordinary growth in the demand for data center and cloud services, fueled by rapid changes in new use cases, application architectures, and hardware technologies, poses substantial challenges to infrastructure systems designers. It takes far too long to understand, integrate, and build the solutions needed to deliver on the potential provided by the buildout of data center and cloud computing capacity. Consider, for instance, how long it takes to exploit the strengths (and avoid the weaknesses) of new ML accelerators, or how long to completely move a complex on-prem application or data flow into the cloud while meeting requirements for availability, data security, and tail performance.

These are not unsolvable problems. Given enough time, and a large enough research and development budget, the hyperscalers have demonstrated that it is possible to integrate new use cases and technologies into existing systems. However, the rate at which we can do these tasks imposes a speed limit on how effectively we can leverage the trillions of dollars of investment in data center infrastructure. The high level of investment needed to optimize workloads also limits the types of workloads that can be supported well.

Our goal is over an order of magnitude improvement in the agility of infrastructure development, so that the Time to Integrate (TTI) for new use cases and technology integration efforts take days or weeks, instead of months and years. Our approach, Self Defining Systems (SDS) is AI native, aimed at leveraging the unique abilities of LLMs to accelerate infrastructure agility, while compensating and masking their weaknesses. Although the focus of the project is on infrastructure, we believe our approach can also dramatically accelerate the rate of innovation by systems researchers and workforce training.

Agentic AI offers a different scaling law for systems development—what we call *scalable agency*. While human engineering capacity grows sub-linearly with headcount [3], agentic workforces can in theory expand nearly instantaneously with available compute. This elastic agentic capacity enables thousands of design and integration hypotheses to be explored in parallel, provided we can also scalably compensate for their weaknesses. Our research agenda is to enable scalable agency to match the pace of hardware and workload evolution without being bottlenecked by human availability.

Consider what is needed to move a complex enterprise application into the cloud. One might start with a high level English description of system components and objectives. These need to be mapped into a formal specification, such as Terraform, and then translated into specific choices as to where and what specific cloud resources to select to meet objectives. A level below, configuring the load balancer for some specific critical element of the application requires understanding the workload, the configuration options of off-the-shelf tools, and the option of introducing application-specific work assignment for greater efficiency or failure resilience. At every level, expertise is needed to generate plausible options, to instrument application data flows to gather evidence to sort through and evaluate those options, to model the likely effect of server and network failures and the performance interference from multiplexed resources, to generate code to put decisions into effect, and to add monitoring to ensure that the application continues to meet its goals after deployment. Of course, this is just one example; similar challenges arise for LLM runtimes, microservices orchestration, scientific applications that use custom accelerators, and data centers for specialized workloads.

LLM-powered agents are both a use case and a potential solution. LLMs offer cheap (relative to a human) but buggy and approximate hypothesis and code generation, equivalent to having an army of inexpensive, tireless, and somewhat clueless junior developers. Industry already knows how to reduce the error rate of junior developers by limiting the scope and complexity of any assigned task. In places, existing systems are already designed with limiting interfaces, such as the use of Linux eBPF or sidecars in microservice architectures. In networking, we often rely on protocol encapsulation to avoid or isolate feature interactions, despite the added overhead. In most technology integration efforts, however, we are forced to rely on expert experience, judgment, and code reviews to bridge gaps and avoid corner case behavior, with the unintended consequence of slowing down technology adaptation.

How do we build data center and shared cloud infrastructure for extreme rates of change? This means designing systems from the ground up for composability, simplicity, and rapid change, while preserving the key benefits of data center and cloud computing: security, reliability, manageability, and resource sharing remain first order considerations. To make this vision a reality, we need advances on a number of fronts; our approach is to deep dive into multiple case studies in parallel, abstracting back to common themes and design patterns. We also plan to proceed from simpler to more complex tasks, with an intermediate goal of producing useful tools for today's practitioners, researchers, and students as we build towards the longer term vision. Examples of needed infrastructure include specification languages for automatic validation of LLM generated code, systems architectures that support the safe insertion of small amounts of measurement and optimization code, simulation frameworks for offline hypothesis testing and anticipating edge behavior, as well as multi-agent systems for automating the hypothesis, code, experiment, and analysis loop.

Recent work has shown that AI can help discover and implement better heuristics for core algorithmic components of complex systems [21, 5, 8]. We take inspiration from these successes and argue for developing entire systems, starting from their high-level specification, and also closing the loop with their deployment and operations. We have recently developed some relevant prototypes, such as specification languages and optimizing compilers for microservices and approximate performance models for tail latency and energy consumption [41, 15, 22], but there is much more to be done. In our view, the only thing that can keep pace with the rate of change of AI is AI itself—to make the integration and troubleshooting of new technologies automatic, with human ingenuity only involved to set up an architecture capable of leveraging scalable and automated exploration of the systems design space.

2 Example Use Case: Self-Defining LLM Runtime

To illustrate the challenges that Self-Defining Systems (SDS) aim to address, consider the problem of building and maintaining an LLM inference runtime—the software layer that manages GPU memory, orchestrates batching and caching, and coordinates requests across heterogeneous accelerators. This layer evolves at a remarkable pace: every few weeks, new attention mechanisms, kernel fusion strategies, or distributed inference methods emerge, each requiring deep manual integration into the runtime. The process of adapting to these changes remains labor-intensive, brittle, and dependent on scarce expert knowledge.

The key difficulty is the growing mismatch between the rate of innovation in AI models and hardware, and the capacity of humans to reengineer complex infrastructure in response. Modern runtimes must balance multiple, often conflicting goals: maximizing throughput while respecting GPU memory limits, avoiding contention across concurrent users, and ensuring reproducibility and correctness under changing models and workloads. Even small design adjustments, such as modifying KV-cache layout or tuning the batch size, can cascade into performance regressions or unpredictable interference across concurrent tenants. Humans spend enormous effort exploring design options, debugging solutions, and validating improvements.

SDS reframes this challenge by treating the runtime as an adaptive system that proposes, evaluates, and refines itself. AI agents hypothesize design variants, generate code, conduct experiments, and analyze results, based on high-level goals. Using a mix of simulation, live evaluation, and A/B testing, they explore a broad design space automatically, identifying tradeoffs and converging toward systems that meet specified goals. Over time, the runtime learns how to evolve alongside changing models and hardware, compressing the TTI for new ideas from months to days.

3 Overview

Figure 1 shows an overview of the SDS approach for our self-defining LLM inference runtime. SDS first iterates with the user to get an appropriate spec of the system. The initial user input will likely be ambiguous or incomplete. SDS helps the user make it clear and complete, which includes external APIs (e.g., POSIX) the system can invoke. SDS then successively refines the spec toward a runnable system. The first stage refines the spec to system architecture which outlines the modules of the system and how they connect. Its output is a document similar to the one that system architects produce today with module interfaces. The second stage refines the architecture document and adds algorithms used by various modules to implement their interfaces. Its output is a set of design documents similar to those that senior engineers produce today. The final stage refines the design documents to runnable code. The artifacts produced by each stage are reviewable by humans. Initially, before the SDS technology matures, we expect humans to review and possibly even edit the artifacts output by each stage so that the deployed system is exactly what they want.

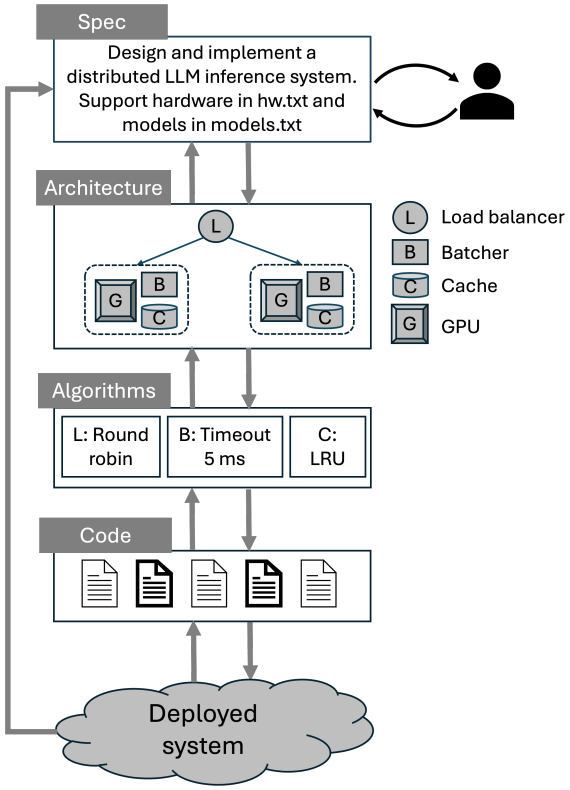

Figure 1: Stages of the SDS workflow using the LLM runtime example. Arrows indicate direct information flow between stages. Execution at a stage is informed by both input from the upstream stage and experience at the downstream stage. Deployment experience informs spec updates as well.

Each stage has multiple possibilities. For complex systems, there tend to be multiple architectures with different modules that can achieve the same spec. Similarly, for a given module there can be multiple algorithmic choices, each with different implications for performance and other system-level metrics of interest. Finally, there are multiple ways to implement a combination of architecture and algorithmic choices. One could do a microservices-like implementation, where modules communicate via RPCs (remote procedure calls), or a monolithic implementation where all modules share a binary.

SDS agents hypothesize refinements at each stage and evaluate the resulting artifacts. We discuss hypothesis generation and evaluation in more detail later, but mention here a few facets of the evaluation process. (1) evaluating the artifacts at upstream stages is more resource efficient but coarser (e.g., architecture evaluation cannot benchmark performance, which depends on algorithmic choices and code as well)—the goal of upstream evaluation is to rule out non-viable options early; and (2) evaluation at a refinement stage informs future hypotheses at that stage as well as upstream stages (backtracking). These two facets mirror how humans design systems in an iterative, feedback-driven manner. Architectures that are deemed poor are not fleshed out; and if an architecture that appeared promising earlier in the process later turns out to be too complex to implement, it is modified or discarded. (3) Because each refinement stage can generate many independent hypotheses, SDS dynamically scales its population of reasoning agents to match the complexity of the design space. During early exploration, agentic capacity expands to cover diverse alternatives; during convergence, it contracts to focus resources on promising configurations. The result is analogous to autoscaling in cloud systems, but applied to reasoning rather than compute.

SDS agents also close the loop on operations and maintenance. They observe how well the system is performing in practice, compare against expectations developed as part of the evaluation, and look for opportunities for improvement. They may realize, for instance, that most requests are short-lived and are

better served by a different algorithm. Based on such observation, it may update the spec and then regenerate code. Spec updates (e.g., supporting a new accelerator) may come from the user as well.

The spec and intermediate artifacts are kept in sync with the running code, and all system updates start with spec changes. This helps lower TTI for new components. Today, when adding a new component, the bulk of the effort is not in writing new code for the component, but in ensuring that code changes do not cause collateral damage. Starting from the spec and current artifacts and re-running the same evaluation make it easy to make such judgments. It also makes it possible to generate small changes to those artifacts that meet the new spec, which lowers risk and makes the changes reviewable by humans. Of course, there is nothing stopping users from providing code changes to describe spec changes. In that case, the system interprets the modified code as a proposed update to the specification and regenerates the intermediate artifacts to maintain consistency.

The multi-stage refinement and closed-loop improvement sets SDS apart from prior AI-driven system design efforts (whose insights we build upon). Prior efforts like AlphaEvolve and ADRS [21, 5] focus on generating code for a particular module, given its interface and an evaluation function. AI is not provided the context within which the module operates, which means that humans must still create and optimize the system architecture and module interfaces. The SDS approach can be applied not just to new systems but to existing ones as well. For applying it to existing systems, where only code may exist, we can use AI to infer the intermediate artifacts and spec, and then handover system operation and updates to SDS agents.

4 A Path from Here to There

We envision a five-phase progression that transforms agents from passive optimizers into autonomous system architects and operators. While we ground the roadmap below in LLM inference runtimes, the same pattern generalizes to other systems. Each phase represents a deeper level of autonomy and reasoning capability, progressing from low-level tuning to meta-level self-improvement. Together they form successive stages in the evolution of a self-defining system.

Phase 1: Self-configuring. In this first phase, the agent is provided with the system model and must tune parameters to meet objectives such as throughput or latency. It manipulates configuration values, simulates and experiments, and reports optimal configurations. For the LLM runtime, this means adjusting batch sizes, KV cache parameters, and memory pool sizes under varying workloads and GPU memory limits. Evaluation metrics for this phase include reduction in human tuning effort, fraction of automatically validated configurations, and performance parity or improvement versus expert baselines.

Phase 2: Self-assembling. After mastering configuration, the agent learns to compose architectures from modular components. It receives a library of schedulers, cache strategies, and kernel designs and must select, interconnect, and tune them to achieve a goal. For example, it may pair a specific KV cache design with a priority-aware scheduler and test the design in simulation.

This phase expands the design space while retaining safety through modularity. The self-configuring capabilities of Phase 1 are used to tune each chosen module. Evaluation metrics for this phase include breadth of design-space coverage within a fixed compute budget, aggregate performance improvement, and cross-component compatibility and reuse across runs.

Phase 3: Self-creating. In this phase, the agent gains creative autonomy: it may propose and implement new runtime components or abstractions. Using evolutionary or reinforcement-learning approaches, the agent generates and evaluates hypotheses. For example, it might invent a hybrid KV-cache that blends prefix- and embedding-based reuse, synthesize the code, and benchmark its performance. Evaluation metrics for this phase include novelty and diversity of validated designs, speed of convergence to Pareto-optimal configurations, and improvement over prior human or agent-generated versions.

Phase 4: Self-designing. The previous phases can evolve existing systems. This phase allows starting from scratch. Before an agent can design a new system, it must be able to describe it accurately. In this phase, the agent analyzes the user-provided spec, potentially referring to related, existing runtimes (e.g., their codebase, execution traces, and configuration files) to infer a machine-readable model of the system: components, dependencies, and data flows. For an LLM runtime, this includes identifying the relationships between scheduling, memory allocation, kernel execution, and cache management.

This phase establishes the declarative foundation that earlier phases can use to build and deploy the system description. The agent produces an internal representation that captures structure and constraints. Evaluation metrics for this phase include fidelity of the extracted model compared to ground truth, completeness of component graph, and correctness of dependency reasoning.

Phase 5: Self-managing. Here, the agents not only design new systems but also improve their own design methodology. It meta-learns which prompts, scaffolds, and feedback signals accelerate convergence. This phase closes the loop: design processes themselves become adaptive artifacts. In the LLM-runtime example, the agent might discover that structured feedback from simulation traces yields faster improvements than raw benchmark scores and update its workflow accordingly. Evaluation metrics for this phase include rate of improvement in design iteration efficiency, stability of learned scaffolds across diverse runtime families, and reduction in overall time to integrate (TTI) for new ideas.

5 Research Challenges

When creating practical self-defining systems, there are challenges at each of the stages in Figure 1 and at each of the phases progressing to a self-managing system. We outline some of those issues here.

All: Agentic workflow. A key research challenge is determining the agentic workflow: which agents do we need at each layer, and how do they fit together. This is similar to how we might construct a software architecture to modularize and isolate concerns between cooperating developers and developer teams. The workflow also has dependencies across layers, similar to how a product manager might call on the expertise in the engineering team in order to conduct a cost-benefit analysis for feature prioritization. A unique research challenge for SDS is scalability; unlike fixed human teams, the number of active agents can fluctuate rapidly as exploration load changes. The workflow must support distributed context sharing, decentralized evaluation, and consistent decision-making across an expanding and contracting agent population. We must architect control loops that allow thousands of agents to explore, learn, and converge in parallel.

Likewise, we want the workflow to enable agents to improve over time. Both human engineers and computational models can be much more accurate and effective if they calibrate themselves against real-world experience versus run open loop. This learning could be from human expertise, when there is a human in the loop, or as a result of later stages that validate and evaluate an approach. At any stage of the workflow, we can ask each agent to produce both a strategy and an estimate or prediction for how well that strategy will work when filtered through downstream stages of the pipeline. What is an effective architecture for agent-to-agent cooperation and mutual learning?

Specification: Formalizing the objective. AI agents work best with a clearly articulated goal, but systems development is inherently multifaceted. As David Patterson once said, "Be careful what you choose as a benchmark, as people [or AI agents] will optimize for it." The systems research community has long had an intense focus on performance to the exclusion of all other concerns; a recent study showed that a third of systems research papers are solely concerned with performance [5]. Yet simple performance benchmarks can be deceiving, as performance gains are only relevant if they do not jeopardize other, more important concerns. Systems developers need to balance a large number of competing concerns: reliability, security, manageability, observability, agility, cost, resource efficiency, energy usage, in addition to performance [31]. Much of the expertise that humans bring to system design is judgment as to how to make these tradeoffs. How does one specify the goal such that agents can produce an artifact that balances real-world objectives?

Architecture: Hypothesis generation. Without taking a stand on the ongoing debate in the AI community as to whether LLMs are stochastic mimics or can create wholly new ideas, we observe that much of the progress in systems research and practice is from some new combination of known techniques. This is a strength, rather than a weakness, of the SDS approach. The sheer number of innovative systems research ideas published each year is far beyond what a human could master, but it is far less than what an LLM can master. The systematic, automated, and inexpensive exploration of combinations of ideas provides a practical way forward for the systems community. Of course, to be successful, we need to be able to encourage or fine tune LLMs to identify and then operationalize combinations of ideas from the literature. Can we use AI to automatically generate architecturally-sound combinations?

Architecture: Component interactions. The architecture defines what components to combine, and what additional algorithms and code modules are needed to accomplish the specification. However, characterizing the semantics or the performance of existing components is a black art. Of course, humans also struggle when making changes that interact with existing code in subtle ways. A key step in using some component will be to systematically experiment (or if the source code is available, systematically analyze) to characterize its behavior in ways important to the target system that might use it. This is a labor intensive step that could be automated. Can we use agents to learn to avoid pitfalls when assembling and configuring existing components?

Algorithms: Use-case specialization. Human systems designers are taught to optimize for the common case, but this can make systems fragile to changing workload assumptions. The speed of algorithm evolution in AI makes this tendency particularly problematic for systems developers. By dramatically improving software productivity, AI itself can help expand the scope of self-defining systems. We are moving into an era where the life cycle cost of developing new software is much lower. Custom solutions for the long tail of use cases are often ruled out today as being prohibitively expensive, and as a result, design constraints on system evolution is often limited by the needs to balance costs and benefits of different users. Can AI-generated specialization slice the Gordian knot of general-purpose systems software?

All: Efficient hypothesis rejection. To make self-defining systems a reality, we must be able to rapidly generate, evaluate, and disqualify decisions at each stage. AI methods generate much of their power through parallel search against defined metrics, rather than encapsulating well-earned expertise. As a result, most hypotheses will be invalid, and those need to be rejected quickly and efficiently, without human intervention. High fidelity evaluation of changes is the gold standard today, but that can both extremely expensive and disruptive to users if it involves live deployment. We thus need an ability to evaluate hypotheses at multiple levels of abstraction, as we progress from specification to architecture to algorithm selection and code.

In places, this might mean generating a rapid prototype for the purpose of aiding an initial evaluation; the prototype code attempts to quantify or validate key assumptions while leaving optimization for later, if the assumptions prove valid. In other words, the agentic loop can rely on the agentic loop itself for answering questions needed at some stage. This is a common technique with human engineering teams, and we believe it will become even more important as AI reduces the cost of software development.

As another example, recent work (by ourselves and others) has shown that AI-assisted models can provide fast, approximate answers, calibrated by feedback from real deployments. While these methods can improve human research productivity, we believe they can supercharge the effectiveness of self-defining systems. Can we efficiently evaluate alternative approaches, along every evaluation dimension that matters in practice, so that only the most promising approaches require live deployment on real workloads?

All: Security/robustness analysis and verification. For people to be comfortable with relying on a self-defining system, the agentic workflow would need to analyze worst case behavior and provide a convincing argument that the behavior of the system will be acceptable if deployed. How will the system perform if x% of its replicas fail or an aggregation switch goes down? Can the system deadlock under adverse conditions? Is its behavior stable under extreme overload conditions? Under what conditions will the system lose data?

These questions are even more difficult, and even more important, to answer for mission- and security-critical systems—where an adversary is trying its best to disrupt operations, and any breach can be catastrophic.

Data center operators spend enormous effort at anticipating and addressing edge case behavior, failure resilience, and security vulnerabilities, with remarkable success compared to the state of the art a decade ago. Even so, human code review and edge case analysis can be error prone. This has driven an increasing interest in the use of formal methods as a complement to more manual methods, to catch more of these types of errors before deployment. Indeed, various projects of our own have pushed the state of the art in formal methods for network configuration and distributed systems [4, 33]. Although fully automated verification for concurrent multiprocessor or distributed networked systems remains a grand challenge, lighter weight formal methods such as model checking have proven effective at reducing bugs and vulnerabilities in complex systems code [2]. These involve developing digital twins to validate functional correctness, as well as advanced model checking frameworks for locating Heisenbugs due to concurrency and device failures. (A concurrency bug was the root cause for Amazon's recent multi-hour outage.) Many of these techniques rely on complex engineering artifacts. Can we use AI to help generate the infrastructure needed to reduce edge case errors?

Deployment: Incremental update. When specifications change, one approach is to re-derive a new system that meets the revised specifications. Depending on how extensive the change is, a more efficient and often more reliable approach is to reuse prior work whenever possible. In human engineering teams, the rate at which we can make changes to existing systems is gated by how frequently those changes disrupt ongoing operations [1]. Since our approach aims to accelerate the pace of technology adoption, an SDS should attempt to leverage deployment experience as much as possible. As a simple example, in a self-configuring system, we might limit ourselves to deltas from previously deployed settings. More broadly, changes must be carefully staged and chosen with an eye to maintaining backward compatibility, performing incremental validation, and the potential for future changes to requirements. If making some change improves some metric, but puts the system into technical or operational debt, then it is almost certainly not worth it. Even if the objective hasn't changed, incremental changes may still be needed, e.g., in response to changes in workload, changes in hardware availability or cost, or even changes in the knowledge base (e.g., if some research suggests a better solution to some aspect of the system, it may change design tradeoffs and therefore outcomes).

All: Knowing when to move on. Human-engineered systems typically operate open loop—continue to refine and improve the system in the most cost-effective manner possible, until higher level management tells you to move on to some (now) more important task. Because a self-defining system will operate at a pace much faster than a human development team, we will need the system to monitor its own progress—to notice when it has reached the stage of diminishing returns, or has become stuck because the most effective plan can't be implemented. And by lowering the cost of development, we make it cheaper to keep going. How do we build an agentic pipeline that can monitor itself?

6 Ongoing Work: Self-Defining LLM Runtime Case Study

As a case study, we are building the LLM inference runtime described in Section 2 from scratch, using AI coding assistants for all design and implementation. Human involvement focuses on setting goals, structuring tasks, and evaluating outputs. Our motivating question is: *How effective are LLM-based agents at realizing the SDS workflow when constructing and optimizing complex systems components?* To this end, we investigate solutions to the research challenges outlined in Section 5. In particular, we ask:

1. How should we structure the agentic workflow—roles, context sharing, and control loops—to scale exploration while keeping human oversight small? How can the workflow detect diminishing returns, i.e., how would it know when to move on? We measure time-to-first-correct-build, total iteration cost, and the frequency and nature of required human interventions.
2. How does the form and completeness of the specification affect correctness, performance, and convergence speed? We evaluate specifications ranging from high-level goals to structured prompts with explicit constraints and regression tests.

3. How well do agents generate architecturally sound hypotheses, and how well do they anticipate subtle component interactions that can induce regressions? We compare different systems organizations (e.g., modular versus monolithic) and assess system stability, development progress rate, and code reuse.
4. How can we optimize the workflow to reject bad hypotheses early, using multi-level evaluation (cheap early signals vs. expensive, high-fidelity tests)? How robust are the resulting systems under failures and adversarial conditions, and what verification/guardrails mitigate risk? Finally, when updating working systems, how well can agents produce safe, incremental *deployment deltas* without destabilizing operations?

Methodology. Inspired by AlphaEvolve [21], we build an iterative hypothesis-evaluate-refine loop over the multi-stage SDS workflow outlined in Section 3. Starting from a user-facing specification, agents propose candidate refinements at each stage and we evaluate the resulting artifacts with increasing fidelity as we move downstream. Upstream evaluation is cheaper but necessarily coarser (aimed at rejecting non-viable options early), while downstream evaluation can trigger backtracking when later results reveal earlier design mistakes or infeasible choices. Our current methodology retains goal setting, architecture decomposition, and evaluation design as human responsibilities, while agents take over much of the implementation and optimization work. Further work will be required to realize the full SDS vision. The resulting workflow lets us measure (i) how different specifications and workflow structures affect TTI and iteration cost and (ii) how well agents generate and select architectural/algorithmic hypotheses.

Early results. So far, our case study has produced two working LLM runtimes, `allmos_v2` and a `monolithic-llm-runtime`. Together, they let us quantify how far AI coding agents can go today. To do so, we evaluate all systems on a throughput benchmark that runs a single Qwen3-0.6B LLM model on one NVIDIA L4 GPU in a Google Cloud Platform (GCP) VM, using short prompts and low-concurrency workloads, reporting steady-state token throughput (tokens/s). As a performance baseline, we use nano-vLLM [6], a simple but high performance LLM runtime that is compatible with and provides similar performance to the popular vLLM [13] runtime. Our throughput benchmark is what the nano-vLLM authors used to demonstrate nano-vLLM's performance parity with vLLM.

- To build `allmos_v2`, we instructed the agent to base its design on Allmos [7], a rudimentary (and slow) LLM runtime that we built using ChatGPT. We further instructed the agent to construct `allmos_v2` to reach the performance of nano-vLLM. `allmos_v2` reaches roughly 1.7k tokens/s, effectively matching nano-vLLM's 1.76k tokens/s while delivering a 76× speedup over the original Allmos baseline.
- We built the `monolithic-llm-runtime` from scratch with a much shorter prompt sequence and less human structure. We intentionally eschewed providing a reference to a baseline system and asked the agent to keep the code simple. `monolithic-llm-runtime` attains 1.2k tokens/s, a 53× speedup since inception.

These prototypes suggest that agents can assemble high-performance systems from scratch with little structure. For both runtimes, the development and optimization path was largely agent-driven: starting from a naïve PyTorch baseline (∼150 tok/s), the agent implemented FlashAttention, CUDA graphs, KV-cache reuse, continuous batching, and prefix caching and each contributed substantial incremental speedups, with CUDA graphs and KV-cache reuse providing the largest gains. The agent also implemented PagedAttention-style block management and hash-based prefix caching, and learned to profile bottlenecks and reason explicitly about compute versus memory-bandwidth limits.

Over time, we evolved from "agent as code generator" to near-autonomous implement-deploy-benchmark loops: Our agent now routinely performs Git operations, provisions and configures new GCP VMs, installs drivers and dependencies, runs benchmarks, and summarizes results. A notable milestone was an agent-authored "key solutions" document that distilled past deployment failures (GLIBC mismatches, missing CUDA toolkits, driver issues) into a reusable playbook, cutting end-to-end deployment from ∼90 minutes of human-in-the-loop debugging to ∼6 minutes with no human intervention on fresh VMs. This allowed us to accelerate the development process from 35 days for `allmos_v2` to just 2 days for `monolithic-llm-runtime`.

However, so far, our agent failed to exceed nano-vLLM's performance. It unsuccessfully attempted FP8 quantization, chunked prefill, `torch.compile`, and custom Triton kernels. In these cases our agent rediscovered techniques already subsumed by CUDA graphs or ran into subtle incompatibilities and workload

mismatches, plateauing without proposing qualitatively new designs (e.g., speculative decoding or fundamentally different kernel structures). We are currently experimenting if explicitly asking the agents to consider the broader systems literature can overcome these humps.

7 Additional Use Cases

We now describe additional use cases where the SDS approach can substantially increase agility and performance or unlock operational modalities that are not possible today. They share the traits that their operational environment (workload, available hardware, new software capabilities, etc.) is fast-paced and the TTI is high, which makes them difficult to develop or operate optimally using current methods. We plan to pursue some of these use cases as we develop an SDS workflow that works for a broad range of systems.

Microservices management. Microservices are the dominant paradigm for developing and deploying distributed applications. Application functionality is decomposed into multiple services, each deployed inside an independent container. Today, a container orchestrator such as Kubernetes deploys and scales microservices and a service mesh such as Istio implements communication policy between microservices. Given the high cost of developing and modifying such systems today, most developers are forced to use these general-purpose systems without the ability to adapt them to their use cases (e.g., the same system is used for long-running microservices and for serverless environments). This imposes a high runtime cost, sometimes as high as 2x [42], for applications. SDS has the potential to dramatically reduce the cost of creating microservices management systems optimized for specific environments. We have in the past developed such systems using the traditional approach [41], which provides a comparative benchmark for SDS.

Deploying complex applications in the cloud. The easiest way to consume computing today is using public clouds which provide a rich array of first- and third-party virtualized services such as Web servers, load balancers, firewalls, access control, cluster orchestrators, and so on. While infrastructure-as-code (IaC) technologies such as Terraform [9] make it easy to consume these services, infrastructure engineers are left to their own devices when it comes to combining these services into an infrastructure layer that can run their collection of applications in a secure, reliable, and performant manner. Unlike physical infrastructure, which evolves slowly once deployed, virtual infrastructure can be and needs to be updated frequently to take full advantage of new services and to remove deprecated services. SDS can enable companies to quickly create frameworks to optimally deploy and operate complex applications atop public cloud services.

Scientific computing. The performance of scientific computing relies on algorithms, hardware, and the systems software (OS, compilers, task schedulers, file systems, network protocols, etc.). Recent advances in AI are helping discover ever faster algorithms, and decreases in the development cost of custom silicon has meant that a plethora of new accelerators are available or custom ones can be developed. What stands in the way of rapid progress is systems software to manage and run scientific applications atop available hardware. Unless optimized, the performance is fragile, so scientists spend enormous effort optimizing for the cross product of specific applications, computer architectures, and network designs. Applications are deeply parallel, with performance gated not just by single core performance but also by all the other components in the system.

Many science teams do not have the necessary resources and expertise, which leaves a wide gap between what is possible and what is achievable. Hyperscalers have the resources and expertise, but the level of investment needed means that they are unable to optimize for workloads in the tail (from a return on investment perspective) where most scientific computation lies. SDS can enable science teams to develop and operate systems optimized for their applications, at a fraction of the previous effort.

Specialized data centers. The public cloud is often not the most cost-effective or best-optimized option for mature workloads [32] (e.g., not every application that uses S3 needs its costly 11 nines of availability). Companies still use the public cloud because they lack expertise to develop systems optimized for their

workload and to operate them with high reliability. SDS can help such companies build custom, cost-effective data centers that seamlessly expand to the public cloud as needed for elasticity and specific services.

8 Related Work

Our work builds on a rapidly expanding body of research at the intersection of AI-assisted scientific discovery, systems design, and agentic automation. These efforts reveal both the potential and the current limitations of applying AI to open-ended design spaces.

AI for discovery and program search. Early milestones such as AlphaGo [25], AlphaGo Zero [26], and AlphaFold [12] demonstrate that deep reinforcement learning can master domains long thought to require human intuition. More recent work, including AlphaDev [18], which discovered faster sorting algorithms, and FunSearch [24], which uses LLMs to discover mathematical results, shows that AI can operate directly on symbolic and programmatic substrates. AlphaEvolve [21] extends this paradigm to code evolution itself, using evolutionary loops over LLM-generated programs to improve scientific algorithms autonomously. Collectively, these systems establish that self-improvement through iterative generation, evaluation, and selection is feasible once a clear objective function and simulation environment are available.

A second thread of work explores AI as a participant in the research process. MLGym [20] and Code Researcher [27] exemplify increasingly capable research agents that can generate hypotheses, design experiments, and analyze results from large codebases. These systems demonstrate that LLMs can operate as autonomous collaborators within bounded scientific and engineering tasks. SDS extends this idea to system-level reasoning, where the architecture, algorithms, and code co-evolve.

Industry efforts such as GitHub's Spec-Driven Development [30] highlight a movement toward declarative, verifiable, AI-assisted engineering. SDS generalizes this philosophy beyond individual applications: rather than using AI merely to fill in code from a specification, SDS treats the entire system as a self-evolving loop.

Formalization and specification. A key challenge, highlighted in work on autoformalization [34] and formal engineering specifications [28], is that most current AI systems lack precise formal targets. Efforts such as Autonomous Code Evolution [36] and TextGrad [37] explore ways to make reasoning and optimization differentiable through text, bridging natural-language prompts and formal objectives. SDS builds on specification-driven AI in which system goals and constraints are explicit and machine-checkable.

AI-driven system design. Work on AI-native infrastructure automation and AI-driven systems provides direct precursors to SDS. Chip Placement with Deep Reinforcement Learning [19], AlphaGo Moment for Model Architecture Discovery [17], and Automated Design of Agentic Systems [10] apply AI to optimize or invent specific architectures under fixed objectives and human-defined design spaces. The Darwin Gödel Machine [38] pushes further toward self-improvement through recursive meta-learning, while Barbarians at the Gate [5] surveys how such approaches are beginning to reshape systems research itself. Most relevant to SDS, Glia [8] introduces a human-inspired, LLM-based multi-agent workflow that autonomously designs mechanisms for LLM-serving clusters. Glia demonstrates that experiment-in-the-loop agentic systems can generate interpretable scheduling and resource management algorithms that rival human designs. Within infrastructure, iServe [16] and AI Native Infrastructure Automation [11] explore intent-based and adaptive control planes. Unlike these systems, which target specific mechanism-level design problems within fixed architectures, SDS aims to enable end-to-end system self-definition. SDS generalizes beyond discovering or tuning components to inferring specifications, generating and evaluating hypotheses across specification, architecture, algorithms, and code, and evolving both design and methodology over time in response to changing workloads, hardware, and objectives.

A foundational intellectual backdrop for SDS is Sutton's Bitter Lesson [29], which argues that across decades of AI research, methods that leverage general computation and learning ultimately surpass those that rely on human insight and manual design. Sutton's motivation was epistemic—the recognition that AI systems will, in the long run, outperform humans at producing better solutions. The SDS vision, by contrast, is

pragmatic: it arises from the growing mismatch between the speed of technological evolution and the rate at which human engineers can integrate new capabilities into complex infrastructure. In effect, SDS takes the bitter lesson to heart but redirects it toward the systems domain, seeking not merely to build better AI, but to apply AI's general learning capabilities to the continuous design and adaptation of infrastructure itself.

Simulation and evaluation frameworks. Because direct experimentation on production systems is costly, simulation frameworks are critical for rapid iteration. Phantora [23] proposes a hybrid simulation layer that maximizes code reuse for performance estimation. Recent work by a number of groups, including ours, has examined using machine learning to produce fast, approximate estimates of simulated or emulated system performance [39, 40, 35, 14, 15]. SDS leverages this work, relying on multi-fidelity reasoning, spanning system descriptions, simulation, and live flighting, to guide agentic exploration at scale.

9 Summary

Self-Defining Systems (SDS) envision a future in which infrastructure can design, validate, and evolve itself with minimal human intervention. By embedding AI agents directly into the systems design loop, SDS transforms infrastructure development from a manual, expert-driven process into an iterative, self-improving one. A key enabler of this vision is scalable agency—the ability of AI systems to expand their reasoning and design throughput in proportion to the complexity or urgency of the task. Unlike human teams, which are limited by organizational growth and training time, agentic collectives can instantaneously parallelize exploration across compute resources. The result is a step-change in agility: shrinking the time to integrate new ideas or technologies from months to days, while preserving the reliability, security, and performance guarantees essential to datacenter-scale systems.

The SDS vision of fully automated system design is a north star for us, and we are cognizant of the fact that we may not fully get there in the foreseeable future. But we are excited about exploring it and learning how close we can get; even a partial realization has the potential to unlock massive economic and societal value by making systems development more efficient and more accessible for many workloads.

We have started developing a self-defining LLM runtime, and our early experiments are promising. We plan to start working in parallel on the other use cases from Section 7, allowing us to discover general agentic workflows that work in a range of domains and find effective technical solutions to challenges in Section 5.

Acknowledgments. Vinamra Agarwal and Soham Bhosale contributed to the LLM runtime experiments.

References

[1] Heather Adkins, Betsy Beyer, Paul Blankinship, Ana Oprea, Piotr Lewandowski, and Adam Stubblefield. *Building Secure and Reliable Systems.* O'Reilly, 2020.

[2] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, 2021.

[3] Fred Brooks. *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[4] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 122–135, 2023.

[5] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How AI is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

[6] GeeeekExplorer et al. nano-vllm: Lightweight llm inference engine. https://github.com/GeeeekExplorer/nano-vllm, 2025.

[7] Simon Peter et al. Allmos. https://gitlab.cs.washington.edu/syslab/allmos, 2025.

[8] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A human-inspired ai for automated systems design and optimization. *arXiv preprint arXiv:2510.27176*, 2025.

[9] HashiCorp. Terraform: Automate infrastructure on any cloud. https://developer.hashicorp.com/terraform, 2025. Accessed 12 Nov 2025.

[10] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

[11] Adam Jacob. AI native infrastructure automation. *System Initiative Blog*, 2025. https://systeminit.com/blog/ai-native-infrastructure-automation.

[12] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596:583–589, 2021.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, 2023.

[14] Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. m3: Accurate flow-level performance estimation using machine learning. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 813–827, 2024.

[15] Chenning Li, Anton A. Zabreyko, Arash Nasr-Esfahany, Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas Anderson. m4: A learned flow-level network simulator. *arXiv preprint arXiv:2503.01770*, 2025.

[16] Dimitrios Liakopoulos, Tianrui Hu, Prasoon Sinha, and Neeraja J. Yadwadkar. iServe: An intent-based serving system for LLMs. *arXiv preprint arXiv:2501.13111*, 2025.

[17] Yixiu Liu, Yang Nan, Weixian Xu, Xiangkun Hu, Lyumanshan Ye, Zhen Qin, and Pengfei Liu. AlphaGo moment for model architecture discovery. *arXiv preprint arXiv:2507.18074*, 2025.

[18] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 2023.

[19] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Anand Babu, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*, 2020.

[20] Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ri-

cardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. MLGym: A new framework and benchmark for advancing AI research agents. *arXiv preprint arXiv:2502.14499*, 2025.

[21] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv preprint arXiv:2506.13131, 2025.

[22] Jonggyu Park, Theano Stavrinos, Simon Peter, and Thomas Anderson. EMPower: The case for a cloud power control plane. *SIGENERGY Energy Inform. Rev.*, 4(5):76–83, April 2025.

[23] Jianxing Qin, Jingrong Chen, Xinhao Kong, Yongji Wu, Tianjun Yuan, Liang Luo, Zhaodong Wang, Ying Zhang, Tingjun Chen, Alvin R. Lebeck, and Danyang Zhuo. Phantora: Maximizing code reuse in simulation-based machine learning system performance estimation. *arXiv:2505.01616*, 2025.

[24] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

[25] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[27] Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. Code researcher: Deep research agent for large systems code and commit history. *arXiv preprint arXiv:2506.11060*, 2025.

[28] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. Specifications: The missing link to making the development of LLM systems an engineering discipline. *arXiv preprint arXiv:2412.05299*, 2024.

[29] Richard S. Sutton. The bitter lesson. *Incomplete Ideas Blog*, March 2019. `http://incompleteideas.net/IncIdeas/BitterLesson.html`.

[30] GitHub Engineering Team. Spec-driven development with AI. *GitHub Blog*, 2024. `https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/`.

[31] Amin Vahdat. Networking challenges for the next decade. In *Open Networking Summit (ONS) 2017 Keynote*, 2017. `https://events.static.linuxfound.org/sites/events/files/slides/ONS%20Keynote%20Vahdat%202017.pdf`.

[32] Sarah Wang and Martin Casado. The cost of cloud, a trillion dollar paradox, 2021. `https://a16z.com/the-cost-of-cloud-a-trillion-dollar-paradox/`.

[33] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, June 2015.

[34] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022.

[35] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 441–457, 2022.

[36] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous code evolution meets NP-completeness. *arXiv preprint arXiv:2509.07367*, 2025.

[37] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[38] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

[39] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. MimicNet: Fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 287–304, 2021.

[40] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 685–702, Boston, MA, April 2023.

[41] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. Application defined networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 87–94, 2023.

[42] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, 2023.