

Monolift: Automating Distribution With the Tools You Have at Home

Tim Goodwin
tgoodwi@ucsc.edu

University of California, Santa Cruz
USA

Andi Quinn
aquinn1@ucsc.edu

University of California, Santa Cruz
USA

Esteban Ramos
esiramos@ucsc.edu

University of California, Santa Cruz
USA

Lindsey Kuper
lkuper@ucsc.edu

University of California, Santa Cruz
USA

Abstract

In recent years, the microservice architecture has become the standard technique for building scalable applications, yet it still confronts developers with numerous challenges. Splitting an application into separate pieces can lead to unpredictable performance, and the process of doing so is complex and labor-intensive. Although a variety of solutions exist to simplify distributed application development, their programming abstractions offer poor support for existing applications in need of scaling. In this paper, we propose Monolift, a new technique for developing distributed applications that prioritizes incremental adoption and support for legacy code. Our proposal treats distribution as a compiler pass, allowing users to guide the distribution process with lightweight code-level annotations, and supports dynamic distribution strategies based on performance metrics rather than development-time decisions. We show that our Monolift prototype can automatically synthesize a variety of distributed architectures from the same source code, each having different performance tradeoffs, and that Monolift's dynamic distribution mechanisms outperform static microservice architectures under varying run-time conditions.

CCS Concepts: • Software and its engineering → Cloud computing; Compilers; Development frameworks and environments.

ACM Reference Format:

Tim Goodwin, Esteban Ramos, Andi Quinn, and Lindsey Kuper. 2025. Monolift: Automating Distribution With the Tools You Have at Home. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25)*, October 13–16, 2025, Seoul, Republic of Korea. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3764860.3768327>



This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '25, Seoul, Republic of Korea

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2225-7/2025/10

<https://doi.org/10.1145/3764860.3768327>

1 Introduction

The microservice architecture has become the de facto standard for building scalable cloud applications. This architectural pattern, which involves developing and deploying an application as a collection of independent subcomponents, has a variety of benefits: It can improve resource utilization through fine-grained scaling, increases fault tolerance by limiting the blast radius of failures, and accelerates application development across parallel teams.

However, despite these benefits and a wealth of tools designed to simplify their development [11, 12, 18, 32], microservice architectures are challenging to “get right,” as distributing an application can have unexpected performance consequences. Because microservices conflate an application's development model with its deployment model, developers make significant, upfront decisions about their deployed architecture that are often guided by the application's logical structure or the structure of the developing organization [9] rather than by the application's technical needs. Moreover, distributed architectures impose overheads [1, 24] and introduce backpressure effects [14] whose performance impacts can be hard to predict at development time. Under certain workload conditions, these factors can dominate, causing certain microservice architectures to perform less efficiently than simpler, monolithic ones, sometimes by a significant margin [4, 25].

Various prior works have sought to simplify the microservice development process and help developers navigate the performance and scalability tradeoffs inherent in various distribution strategies. *Multitier* [34] languages and frameworks [8, 10, 16, 29, 33] abstract away distributed plumbing code and combine the components (or “tiers”) of an application in the same program, which are then split into independently deployable artifacts by a compiler. Actor systems [3, 22] and distributed object models [20, 26, 28] abstract away distribution through component frameworks that pair with a run-time layer to manage deployment and configuration.

Although these solutions share a common goal of simplifying distributed application development, they impose

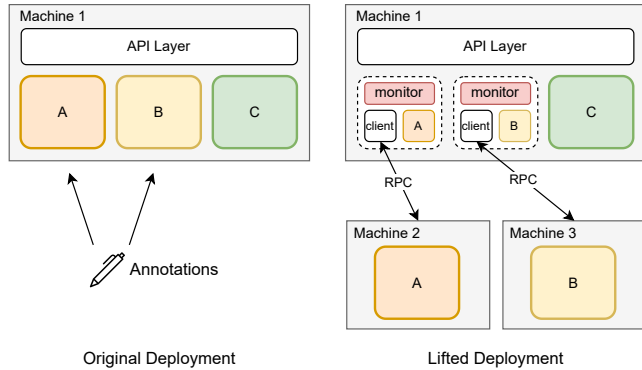


Figure 1. A monolithic application being distributed with Monolift. Program segments A and B have been annotated and transformed into lifts, where dotted regions represent lift points that integrate with a run-time monitor.

“all-or-nothing” programming models that require developers to substantially refactor existing applications to reap their benefits. As application scaling requirements often emerge long after development time, this lack of support for *existing* applications is dire. Google recently released a contribution to this space, Service Weaver [15], only to abandon it a little more than a year later, citing that “we realized that it was hard for users to adopt Service Weaver directly since it required rewriting large parts of existing applications. Therefore, Service Weaver did not see much direct use” [30].

In this paper, we propose Monolift, a new way of writing and deploying distributed applications that supports a “pay-as-you-go” programming model, i.e., developers can gradually employ Monolift techniques as needed to incrementally reap the scalability and performance benefits of distributed architectures.

The key to Monolift’s design is that it treats distribution as a compiler pass for general-purpose programming languages, rather than as a new language, framework, or programming model. For run-time support, Monolift leverages the features of an external container platform by treating it as a compiler target. Distribution is guided by lightweight code-level annotations that can be added incrementally, and are transparent outside of Monolift. Monolift is therefore “low-commitment” in that its distribution is optional and readily reversible; without Monolift, the application works as it originally did. Additionally, Monolift applications can be configured with the ability to dynamically employ distribution based on live performance conditions, enabling developers to ensure that their application’s distribution strategy is based on run-time needs rather than development-time decisions.

The core abstraction in our design is the *lift*, which is an annotated region of code that can run locally or remotely. Distribution unfolds at *lift points*, which are call sites to annotated program segments where computation can branch from

the main program onto remote resources. With lifts, developers can scale arbitrary segments of an application across additional machines, providing a wide variety of strategies to distribute their application’s work without rewriting their code. Because Monolift automates the architectural changes, developers can swiftly explore this design space to find the strategy that works best.

We implement a Monolift prototype for the Go programming language [17] that targets Kubernetes [19] for deployment. We demonstrate that from the same application source, Monolift can generate a variety of distributed architectures, each of which has workload-specific performance characteristics. We then demonstrate how Monolift programs can strategically navigate the performance and scalability trade-offs between various architectural strategies in response to run-time conditions, achieving both the superior latencies of a monolithic application at low loads and the superior scalability of a distributed application as load increases.

In summary, we contribute:

- Monolift, a new technique for developing and deploying distributed applications that supports a “pay-as-you-go” programming paradigm.
- A new approach for dynamically distributing computation that allows developers to specify performance-based policies in a lightweight annotation language.
- An early prototype of Monolift for Go and Kubernetes that enables rapid exploration of the distributed application design space and supports dynamically optimizing distribution to improve performance.

2 The Monolift Vision

We propose Monolift as a new approach to writing and deploying distributed applications. Monolift’s goal is to support a “pay-as-you-go” programming model, meaning that developers can progressively reap the benefits of Monolift’s distribution by incrementally adopting its techniques in an existing monolithic application. In particular, Monolift introduces a new abstraction, called a *lift*, which is a code segment that can run locally or remotely. Developers create lifts in their application by adding lightweight annotations to their code. Our design supports distribution through two components: a compiler pass that transforms a developer-annotated monolithic application into a collection of binaries that support distribution, and a runtime library that distributes the compiled binaries across machines and monitors performance signals to adjust distribution with respect to developer-specified concerns (cost, tail latency, throughput, etc.).

Figure 1 shows an example of the Monolift process. The Monolift compiler converts the original monolithic application (Original Deployment) into a collection of executables that support distribution (Lifted Deployment). The Monolift

compiler embeds the Monolift runtime into the lifted deployment to monitor performance signals and adjust distribution dynamically.

2.1 Lifts

The core abstraction in our design is the lift, which is a code segment that can execute locally or remotely. A lift is a possible unit of distribution, i.e., the Monolift runtime decides which lifts to execute on remote machines based upon performance signals from a deployment. Each lift is associated with an annotated structure in the source program, such as a function or class definition. Because lifts can be created from a variety of program structures, they can take on multiple shapes and sizes, such as a large service module, or a simple background routine that may benefit from offloading. The distributed representation of a lift is determined by the language construct from which it is derived: Monolift distributes functions as ephemeral tasks that are created on demand (e.g., a serverless function), and classes and interface implementations as network services.

Although lifts can execute locally or remotely, they do not migrate [23, 31]. Rather, lifts can *offload* their computation by transferring control to a remote instance of the functionality they encapsulate. In this sense, lifts support a scale-out pattern, where program functionality is replicated onto additional resources to handle load. Figure 1 illustrates how program segments A and B are replicated onto additional machines in the lifted deployment.

A key design decision in Monolift is to *not* support arbitrary lifts. Instead, Monolift supports only the lifts for which it can provide an efficient solution. For example, we do not plan to support lifts that share global state with the rest of the application, since doing so would require using expensive distributed algorithms to synchronize state. Rather, we prioritize a flexible design that provides many opportunities for developers to work within Monolift’s restrictions. For example, a stateful class can be made into a lift that is always invoked remotely on a dedicated service. Users can additionally modify their applications to create lifts for code segments that do not meet Monolift’s requirements, e.g., by modifying functions to take global state as arguments instead of referencing them directly. We anticipate that limiting lift support will be a smart tradeoff since we believe that monolithic applications already include sufficient code segments that meet Monolift’s restrictions and that additional code changes required to support lifts will be minor.

Lift Annotations. To use Monolift, developers create lifts by adding *lift annotations* to their application source. These annotations are applied directly to the language’s constructs for encapsulating program functionality (e.g. function, class, or interface definitions). By associating lifts with these constructs, lift annotations guide the decomposition process along the existing contours of the application’s structure,

enabling developers to leverage the modularity within their code to distribute their applications.

Lift annotations contain *delegate expressions*, which allow developers to specify scenarios in which a lift should be executed locally or remotely. Delegate expressions are embedded within lift annotations using a domain-specific language and can encode static policies (e.g., always invoke the lift remotely), or dynamic distribution policies that can adapt with the application’s workload, depending on local conditions (such as the invocation rate of a given function) or application-wide performance indicators (such as the application’s overall resource consumption). For example, a user could configure a resource-intensive lift to execute remotely if the application’s CPU consumption exceeds a certain threshold.

Users add lift annotations as comments in the source language. Thus, when not compiled with Monolift, the annotated application continues to work as it originally did. This feature is convenient, as it allows developers to explore distribution strategies in a “low-commitment” way and enables them to use existing test suites and development tools designed for centralized software when analyzing their application. Figure 2 illustrates lift annotations being used in our Go prototype of Monolift (see Section 3).

2.2 The Monolift Compiler

During compilation, Monolift transforms an annotated application into a collection of deployment artifacts consisting of the lifted application and an artifact for each lift’s distributed counterpart. The lifted application embeds Monolift’s runtime to dynamically determine whether to execute each lift locally or remotely (see Section 2.3). This approach enables the design to use existing distributed system orchestration platforms such as Kubernetes [19] or Mesos [21] for deployment.

The Monolift compiler performs three main tasks. First, for each lift in the application, the compiler produces a stand-alone binary that includes only the lift’s encapsulated logic and its dependencies. Second, the compiler creates a run-time monitor for each lift that tracks the properties referenced in the lift’s associated lift annotations. Third, the compiler identifies existing call sites to lifted program functionality and converts each one into a *lift point*, which is a wrapper type that encapsulates a network client and handles parameter serialization. In Figure 1, program segments A and B represent lifts, and the dotted-line regions represent lift points. When control flow enters a lift point, it consults with the Monolift runtime to determine whether to invoke the lift locally, using the original code, or to invoke the lift remotely through its network client. Lift points expose the same function or method signatures as the lifts they invoke, so the compiler can replace call sites with lift points without any additional modifications to the surrounding program.

Metric	Identifier	Scope
CPU Utilization (%)	CPU	Per process
Memory Utilization (%)	MEM	Per process
Invocation Rate (req/s)	IPS	Per lift

Table 1. Metrics that can be used with our prototype to configure dynamic distribution strategies.

The end result of Monolift’s compilation process is an application with the same code structure as the original program, except that it includes the Monolift runtime and call sites to annotated code have been replaced with lift points. As all structural changes are encapsulated within internal lift points, the resulting application exposes the same external API as the original application. Therefore, Monolift can transform programs in a manner that is transparent to the application’s external dependencies, enabling developers to freely explore various distribution strategies without worrying about compatibility issues.

2.3 The Monolift Runtime

The Monolift runtime determines whether to execute each lift locally or remotely using the delegate expressions in each lift’s associated lift annotations. Some delegate expressions may specify simple or static policies, while others may specify application-wide performance conditions that the runtime could handle in a variety of ways. We envision that this will work as follows. First, the compiler formulates a state transition model in which each state is a possible configuration of the program’s lifts. Then, it determines a transition function based on the delegate expressions referenced in the program’s lift annotations. Composing the program’s delegate expressions into a global transition function is challenging. For example, suppose the user configures lift A and B in Figure 1 to offload when the application’s CPU utilization reaches 50%. Ideally, the system should offload one of the lifts when the application’s CPU utilization reaches 50%, and then offload the other lift when the already-offloaded application’s CPU utilization reaches 50%, but which should it offload first? We plan to build on profile-guided optimization and learning-based autoscaling [5] to address these challenges.

The Monolift runtime will monitor the application’s performance properties to determine how to execute each lift. It will follow the provided state transition model and transition function, instructing each lift point to handle lift invocations locally or remotely, accordingly.

3 Prototype Implementation

We implement a Monolift prototype for the Go programming language [17] that deploys lifted applications on Kubernetes

```
//monolift:offload metric=CPU threshold=75%
func hashPassword(pw string) (string, error) {
    // resource intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}

//monolift:offload metric=IPS threshold=1000
type UserService interface {
    Register(name, password string) error
}

// An example UserService implementation
type usrMgr struct {
    db database.Client
}

func (u *usrMgr) Register(name, pw string) error {
    if pwHash, err := hashPassword(pw); err != nil {
        return err
    }
    return u.db.Insert("users", name, pwHash)
}
```

Figure 2. Example Go code exhibiting various Monolift offload strategies.

[19]. The prototype supports lift annotations as Go comments and lets users associate lifts with Go function definitions and interface definitions. Additionally, we implement a delegate expression DSL as a key-value interface based on performance signal thresholds. Our prototype DSL currently supports three runtime metrics, shown in Table 1.

Figure 2 shows an example of using the prototype. The example includes two lifts, `UserService` and `hashPassword`, together with two lift annotations, prefixed with the keyword `monolift:offload`.

3.1 Code Extraction

Our prototype compiler parses the ASTs of each Go source file and identifies function and interface definition nodes marked with lift annotations. To extract annotated code into standalone services, the compiler first resolves code dependencies by analyzing how the code is instantiated at the application’s entrypoint. It then reconstructs the instantiation process to ensure that the code includes its dependencies when running as a standalone service. This way, our prototype supports lifts that rely on database clients, loggers, library functions, and configuration parameters. As discussed in Section 2, this extraction process is subject to certain constraints and our prototype will produce an error if annotated code cannot be reliably distributed. The compiler enforces that lift parameters are serializable and that lifted code does not contain heap operations. Additionally, the compiler warns the user if it detects stateful operations or references to package-level variables within annotated

	entrypoint	user	post	timeline	socialgraph
baseline	32	-	-	-	-
full	16	4	4	4	4
post-only	16	-	16	-	-
user-only	16	16	-	-	-
SG-only	16	-	-	-	16
TL-only	16	-	-	16	-

Table 2. Number of CPU cores allocated to each microservice component for each Monolift configuration.

code, as our prototype does not synchronize state between processes in the lifted application.

Once the compiler resolves a lift’s dependencies, it generates a serialization layer to convert function or method parameters to and from a JSON [6] wire format and then encapsulates the extracted code in an HTTP server. For lifted functions, this HTTP server exposes a singular endpoint by which the function is invoked, with the server process exiting after returning a response. For lifted interfaces, the compiler generates an HTTP endpoint for each interface method, and the server runs indefinitely.

3.2 Integration and Deployment

After the extraction process, the compiler generates a client for each extracted service. The compiler then creates lift points for these clients with a wrapper type that exposes the same signature as the lifted functionality, and then updates former call sites with these lift points via AST node substitution. Our prototype leverages the naming conventions of Kubernetes’ internal DNS to determine the addresses of the extracted services at compile time, enabling the resulting collection of artifacts to be readily deployable as a fully integrated application.

4 Evaluation

To evaluate our prototype, we used the Social Network microservice application from DeathStarBench [14]. We ported the existing Go application to a monolithic architecture to serve as a baseline for evaluating Monolift’s decomposition abilities. The Social Network application’s code packages are already organized by application functionality: user, post, timeline, and socialgraph. We defined and implemented Go interfaces for these packages using the existing microservice code, and connected these implementations directly to the application’s API frontend. Our port preserves the service dependencies and invocation semantics of the original architecture. Where the microservice application uses synchronous RPC calls, our interface implementations invoke each other’s respective methods, preserving the same call graph structure. Where the microservice application connects services asynchronously via message queues, we use the same message format, but with in-memory queues.

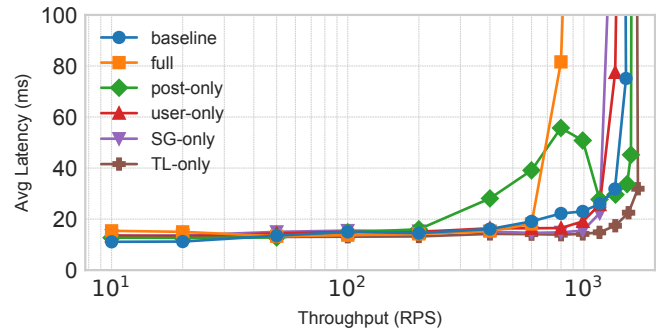


Figure 3. Throughput vs. latency for a variety of compiled Social Network architectures.

We perform all experiments on an 8-node Kubernetes cluster of 40-core servers. Before each experiment, we initialize the application with the Reed98 social graph dataset [14] to simulate realistic application workloads.

4.1 Rapid Prototyping

Our first evaluation goal is to demonstrate how Monolift can be used to quickly explore a variety of possible distributed architectures and the tradeoffs between them. To do so, we apply annotations to generate a collection of distributed architectures, and evaluate their relative performance under a workload consisting of post creation requests. We do not use Monolift’s support for dynamic lift behavior in this experiment, so the resulting configurations represent static microservice architectures. Additionally, we ensure that each compiled architecture is provisioned with the same amount of total resources, and we do not utilize any autoscaling, so that performance differences purely reflect the tradeoffs between the various decomposition strategies.

We evaluate five different annotation placements based on the four code packages of the Social Network application. The first configuration annotates all four interfaces for extraction. The subsequent four configurations extract only the user, post, timeline, or socialgraph interface. In all configurations, each microservice component is deployed on a separate cluster node, and the un-extracted “entrypoint” portion of the application contains the application’s API frontend. We configured static resource allocations for the five resulting architectures, as well as the baseline, which are outlined in Table 2.

The results of our experiment are shown in Figure 3. We observe that traditional distribution techniques, i.e., no distribution (baseline) or fully distributed (full) are suboptimal. Instead, the ideal distribution approach for this workload comes from distributing only the timeline lift. Note that only distributing the timeline lift is likely sub-ideal for other workloads; having the ability to rapidly explore multiple distributed architectures, as provided by the Monolift prototype, is beneficial for exploring these options.

4.2 Dynamic Distribution

Our second evaluation goal is to demonstrate the utility of Monolift’s dynamic distribution mechanisms. Based on the results of our prior experiment, we apply an annotation to the timeline interface with a delegate expression to offload work to a dedicated service if the rate of timeline operations exceeds 100 RPS. We evaluate this Monolift architecture against our monolithic baseline and the original microservice implementation. We deploy the three configurations and conduct an experiment that increases the input load (requests per second) over time.

The tail latencies for each of the deployments over time are shown in Figure 4. We observe that the monolithic deployment (Baseline) has low tail latency when it is capable of handling the input load, but its tail latency spikes once the input load saturates its capacity. In contrast, the microservice design has higher tail latency when at low load but is able to maintain its tail latency when the input load increases. The Monolift configuration provides the best of both worlds: its tail latency is near the baseline in low RPS scenarios, when the input load is low enough that the system does not execute the timeline interface remotely. When the input load increases, the Monolift configuration executes the timeline interface remotely and is able to maintain its tail latency, which is approximately 20% better than that of the microservice configuration.

5 Related Work

Multitier Programming. Our proposal is closely related to the multitier programming paradigm [34]. In the multitier paradigm, developers can write a distributed application as a single program, and a compiler then translates the program into a collection of independently deployable artifacts. Multitier programming originally emerged in the early 2000s as a way to manage the complexity of programming the *tiers* of web applications (for example, the client tier, the server tier, and the database tier), and many multitier languages and frameworks [10, 16, 29] specifically target this domain and assume that applications have a client-server structure. (An exception is ScalaLoc [33], which, like Monolift, supports arbitrary application structure.) Traditional multitier languages and frameworks require the programmer to begin with code written in the language or framework (or port their existing application to it wholesale), whereas Monolift supports a workflow in which programmers can incrementally add annotations to an existing application. Additionally, while multitier frameworks let programmers mix code from different tiers in the same compilation unit, they usually require developers to explicitly specify *where* data and computation should be placed (i.e., on the server or the client) [34]. Even among those multitier frameworks that determine placement automatically, placement is handled by a static, ahead-of-time analysis, whereas Monolift

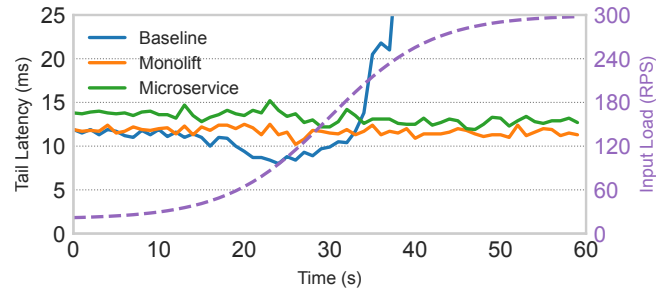


Figure 4. Tail latency for Social Network under an increasing load pattern.

supports a workflow in which placement decisions are made dynamically in response to run-time needs. Finally, multitier programs themselves are not directly executable without the use of a multitier compiler, whereas with Monolift, use of the compiler is always optional.

Actor Systems. Actor systems like Orleans [3] and Akka [22] similarly use abstractions to decouple application structure from run-time configuration and can manage distribution dynamically. However, these systems require use of component frameworks, which impose a specific structure on the applications they support. Monolift achieves abstraction through a language’s existing constructs to support applications of arbitrary structure. Our programming model bears some similarity to Ray [27], a framework for ML applications that supports an actor-like programming model via annotations on ordinary Python functions and classes. The closest solution to our proposal is Service Weaver [15], which shares our vision of running an application either as a single binary or as a distributed system based on workload needs. Like actor systems, Service Weaver employs a component framework, but as a consequence, it offers poor support for incremental adoption.

Offload Frameworks. Our proposal takes some inspiration from programming frameworks that support incrementally offloading computation, either to dedicated hardware or architectural components that are closer to the data being operated on. TELEPORT [36] is one such system that prioritizes programming flexibility by exposing a pushdown syscall to offload certain computation to far memory pools, enabling developers to get more performance out of disaggregated architectures via familiar OS abstractions. Offload Annotations [35] provide a minimally invasive mechanism for developers to integrate established CPU libraries with emerging GPU libraries. Pyxis [7] optimizes performance of conventional database-backed applications by automatically extracting certain application code to stored procedures that run on the database server to minimize data movement.

Container Management Platforms. Systems such as Kubernetes [19] greatly simplify the process of deploying

scalable distributed applications by providing a variety of autoscaling and networking solutions. Our solution leverages these platform features to address *development* challenges by using a compiler that targets container platform APIs. Cluster resource managers [5, 13, 37] optimize performance of containerized application deployments by handling fine grained scaling decisions for individual microservice components. These systems work with applications that have already been adequately decomposed as independently-deployable subcomponents, although their core techniques could be complementary to Monolith's run-time mechanisms.

6 Conclusion and Future Work

In this paper, we presented Monolith, a new technique for developing and deploying distributed applications that prioritizes incremental adoption and support for legacy code. Instead of handling distribution with new abstractions, Monolith treats distribution as a compiler pass for general-purpose languages, lets users guide the distribution process with lightweight code-level annotations, and supports dynamic distribution strategies based on run-time conditions. While the techniques utilized in our design are not fundamentally unique to our proposal, we believe that Monolith composes these techniques into a novel programming model that offers value to developers and may open the door for future innovations.

Additional work will be required to fully realize our vision for Monolith. For instance, our proposed annotation DSL can be used to support powerful dynamic distribution strategies, and we have demonstrated utility, but it will likely be difficult for developers to manually configure annotations for reliably optimal performance outcomes. In general, knowing how and when to distribute a program's computation for optimal performance is a complex problem affected by a multitude of factors [2, 13, 36], and the entirety of this problem space cannot be addressed with a small annotation language. Monolith may therefore benefit from integration with systems that optimize microservice resource allocation for end-to-end performance [5, 13, 37]. These types of systems infer the impact of resource allocation on microservice performance as well as how inter-service relationships affect end-to-end application performance. Monolith's ability to provide a bird's eye view of the distributed applications it produces may make these techniques more powerful, and its lift abstraction may enable new innovations for tuning resource allocations and reducing deployment costs for distributed applications.

Acknowledgments

We would like to thank Peter Alvaro and Achilles Benetopoulos for their insightful feedback on early drafts of this work, as well as the anonymous reviewers for their comments and suggestions that improved the paper. This material is based upon work supported by the National Science Foundation

CSGrad4US Fellowship Program under Grant No. 2240204. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) (*HotOS '19*). Association for Computing Machinery, New York, NY, USA, 113–119. doi:10.1145/3317550.3321434
- [2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) (*EuroSys '20*). Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. doi:10.1145/3342195.3387522
- [3] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Technical Report MSR-TR-2014-41. <https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/>
- [4] Thomas Betts. 2020. To Microservices and Back Again - Why Segment Went Back to a Monolith. <https://www.infoq.com/news/2020/04/microservices-back-again/>. *InfoQ* (27 April 2020). Accessed: 21 July 2025.
- [5] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 623–643. <https://www.usenix.org/conference/osdi23/presentation/bhardwaj>
- [6] T. Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. <https://www.rfc-editor.org/rfc/rfc8259.html>
- [7] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment* 5, 11 (July 2012), 1471–1482. doi:10.14778/2350229.2350262
- [8] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 153–165. doi:10.1145/2676726.2677004
- [9] Melvin E Conway. 1968. How Do Committees Invent? *Datamation* (1968).
- [10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.
- [11] Dapr Project. 2025. Dapr - Distributed Application Runtime. <https://dapr.io/>. Accessed: 2025-08-01.
- [12] Encore Cloud. 2025. DevOps Automation Platform for AWS & GCP – Encore Cloud. <https://encore.cloud/>. Accessed: 2025-08-01.
- [13] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual, USA) (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. doi:10.1145/3445814.3446700

- [14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinisky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 3–18. doi:10.1145/3297858.3304013
- [15] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications (ServiceWeaver). In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 110–117. doi:10.1145/3593856.3595909
- [16] Google. 2006. GWT (Google Web Toolkit). <https://www.gwtproject.org/>. Accessed: 21 July 2025.
- [17] Google. 2009. The Go Programming Language. <https://go.dev/>. Accessed: 21 July 2025.
- [18] Google. 2025. gRPC. <https://grpc.io/>. Accessed: 2025-08-01.
- [19] Google and Cloud Native Computing Foundation. 2014. Kubernetes. <https://kubernetes.io/>. Accessed: 21 July 2025.
- [20] Michi Henning. 2006. The Rise and Fall of CORBA: There's a lot we can learn from CORBA's mistakes. *Queue* 4, 5 (June 2006), 28–34. doi:10.1145/1142031.1142044
- [21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. <https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center>
- [22] Jonas Bonér and Lightbend, Inc. 2009. Akka: A toolkit for building concurrent and distributed applications on the JVM. <https://akka.io/>. Accessed: 21 July 2025.
- [23] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 109–133. doi:10.1145/35037.42182
- [24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, Portland Oregon, 158–169. doi:10.1145/2749469.2750392
- [25] Marcin Kolny. 2023. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. <https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90>. *Prime Video Tech Blog* (22 March 2023). Archived on <https://archive.is/h788m>; Accessed: 21 July 2025.
- [26] Microsoft Corporation. 2023. MS-DCOM: Distributed Component Object Model (DCOM) Remote Protocol. Microsoft Open Specifications. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom/4a893f3d-bd29-48cd-9f43-d9777a4415b0 Last Updated: 25 October 2023; Accessed: 29 July 2025.
- [27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. (2018).
- [28] Oracle Corporation. 2014. Java Remote Method Invocation (RMI). Oracle Java Platform, Standard Edition 8 Documentation. <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html> Accessed: 29 July 2025.
- [29] Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (Nara, Japan) (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 180–192. doi:10.1145/2951913.2951916
- [30] ServiceWeaver Project Contributors. 2024. Service Weaver: A Framework for Building Distributed Applications. <https://github.com/ServiceWeaver/weaver/blob/main/README.md>. Accessed: 21 July 2025.
- [31] Jonathan M. Smith. 1988. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev.* 22, 3 (July 1988), 28–40. doi:10.1145/47671.47673
- [32] Spring Project. 2025. Spring Boot. <https://spring.io/projects/spring-boot>. Accessed: 2025-08-01.
- [33] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoc. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 129:1–129:30. doi:10.1145/3276499
- [34] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *ACM Comput. Surv.* 53, 4, Article 81 (Sept. 2020), 35 pages. doi:10.1145/3397495
- [35] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload annotations: bringing heterogeneous computing to existing libraries and workloads. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '20)*. USENIX Association, USA, Article 20, 14 pages.
- [36] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 1345–1359. doi:10.1145/3514221.3517856
- [37] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 167–181. doi:10.1145/3445814.3446693