Building Efficient Modular Cloud Systems

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Vaastav Anand

Saarbrücken, 2026

Date of Colloquium:	24th September 2026
Dean of Faculty:	Prof. Dr. Jan Reineke
Chair of the Committee:	Prof. Dr. Martina Maggio
Reporters:	Dr. Antoine Kaufmann
	Prof. Dr. Deepak Garg
	Prof. Dr. Tianyin Xu
	Dr. Rebecca Isaacs
Academic Assistant:	Dr. Abhisek Dash

UNIVERSITÄT
DES
SAARLANDES

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, ***2026***

gez. / signed

Vaastav Anand

Abstract

Cloud systems are composed of multiple inter-connected independent subsystems. These systems are complex in nature as they are made of heterogeneous components rife with complicated interactions, operate in dynamic conditions, and exhibit unpredictable behaviors. Developers expend significant manual effort to efficiently design, implement, optimize, operate, and improve these systems throughout the cloud system lifecycle.

This dissertation proposes Human-in-the-loop Specification-Driven Automation (HITLSDA) techniques for reducing the manual effort required by developers for carrying out tasks across the cloud lifecycle. HITLSDA techniques reduce the manual effort by the decoupling the creative component of cloud lifecycle tasks that require human insight from their corresponding mechanical component which can be automated.

We instantiate this HITLSDA approach through two systems targeting different phases of the cloud lifecycle: (i) Blueprint, a toolchain for generating and reconfigurable microservice systems that streamlines exploration of the cloud system design space; and (ii) Iridescent, a framework for online, workload-driven

runtime specialization that continuously optimizes system performance by adapting code to observed execution behavior. Together, these systems demonstrate that HITLSDA provides a general and effective paradigm for reducing developer burden in cloud system design, implementation, and optimization while maintaining flexibility and performance.

Zusammenfassung

Cloud-Systeme bestehen aus vielen miteinander verbundenen, unabhängigen Subsystemen. Aufgrund ihrer heterogenen Komponenten, komplexen Wechselwirkungen, dynamischen Betriebsbedingungen und oft schwer vorhersehbaren Verhaltensweisen sind diese Systeme inhärent komplex. Entwickler müssen daher über den gesamten Lebenszyklus hinweg erheblichen manuellen Aufwand betreiben, um Cloud-Systeme zu entwerfen, zu implementieren, zu optimieren, zu betreiben und weiterzuentwickeln.

Diese Dissertation schlägt Human-in-the-loop Specification-Driven Automation (HITLSDA) als Ansatz zur Reduktion des manuellen Aufwands in Cloud-Lebenszyklusaufgaben vor. Der zentrale Gedanke besteht darin, den kreativen Anteil solcher Aufgaben, der menschliche Expertise erfordert, von dem mechanischen Anteil zu trennen, der automatisiert werden kann.

Dieser Ansatz wird durch zwei Systeme umgesetzt, die unterschiedliche Phasen des Cloud-Lebenszyklus adressieren: (i) Blueprint, eine Toolchain zur Generierung und konfigurierbaren Erstellung von Microservice-Systemen, die die Exploration des Systemdesignraums vereinfacht; und (ii) Iridescent, ein Frame-

work zur laufzeitbasierten, workload-getriebenen Spezialisierung, das Systeme kontinuierlich optimiert, indem es Code an beobachtetes Ausführungsverhalten anpasst. Gemeinsam zeigen diese Systeme, dass HITLSDA ein allgemeines und effektives Paradigma zur Reduktion des Entwicklungsaufwands in Cloud-Systemen darstellt, ohne dabei Flexibilität oder Performance zu beeinträchtigen.

Acknowledgement

I would like to thank my advisors Antoine Kaufmann and Deepak Garg as well as my former advisor Jonathan Mace. Jonathan was my initial advisor (before he left for a researcher position at MSR Redmond). From my very first day as a PhD student in his group, I felt fully supported in all of my PhD escapades. He always listened to and critiqued all of my ideas (good, bad, and ugly) and gave me feedback to improve my ideas and subsequently my research skills. I will always remain indebted to Antoine and Deepak for accepting me as their student when Jonathan left his position at MPI-SWS. They made sure that the transition period between me switching advisors was as smooth as possible. Antoine is the nicest and best advisor one could wish for and his kindness has shown me how to be a successful advisor in the future.

My time at MPI-SWS would not have been such an enjoyable experience without the other systems students: Roberta De Viti, Matheus Stolet, Tejas Harith, Thomas Davidson, Reyhaneh Karimipour, Safya Alzayat, Zhiqiang Xie, Jakob Görgen, Jonas Kaufmann, Marvin Meiers, Pierfrancesco Ingo, Artemio Ageev, and Hejing Li. Also a huge shout out to MPI's FC Saarbruecken!

My UBC Master's Crew — Puneet Mehrotra, Joe Wonsil, Stewart Grant, Chris Chen, Clement Fung, Paulette Koronkevich, Tony Mason, Amanda Levin, and Surbhi Palande played a major role in my academic journey. I would also like to thank my MSc advisors Ivan Beschastnikh and Margo Seltzer for instilling the research foundations in me. They were the first to believe in me and my potential and for that I am eternally grateful.

A huge shoutout to my internship mentors and collaborators at Microsoft: Rodrigo Fonseca, Alok Gautam Kumbhare, Pedro Las-Casas, Gagan Somashekar, Chetan Bansal, Ricardo Bianchini, Jonathan Mace, Celine Irvene. Also a huge thank you to all the other members of the Azure Research — Systems team for creating a welcoming and nurturing environment during my internship. The three months I spent as an intern in the AzRS team were the best three months of my PhD journey.

A special thanks to Justine Kaufmann for her invaluable help in designing logos for my projects and for the professional photographs. And a huge thanks to Tilney and Dalloway for ensuring that my research passed the sniff test.

I would also like to thank my friends Sultan, Milagros, Sonalee, Daniela, Cailey, and Devrat for always being there for me in times of support.

Finally and most importantly, I thank my family for their support. My mother, Shagun, my uncle, Saket, and my grandmother, Daman, have been constant sources of support and encouragement.

Dedication

To my family.

In loving memory of my grandfather, Mr. Satish Kumar Anand.

Table of Contents

List of Figures xx

List of Tables xxii

List of Abbreviations xxiii

I Introduction 1

1 Introduction 2

1.1 Thesis Statement and Goal 4

1.2 Thesis Contributions 5

1.2.1 Human-in-the-loop Specification-Driven Automation . . 5

1.2.2 Two HITLSDA Instances for the Cloud Lifecycle 7

1.2.2.1 Rapid Offline Design Prototyping with Blueprint 7

1.2.2.2 Online Code Specialization of Systems with Iridescent 8

1.3 Publications and Collaborations 9

1.3.1 Thesis Publications 9

1.3.2 Non-Thesis Publications 11

2 A Primer on Modular Cloud Systems 14

2.1 Cloud Systems Structure 14

2.2 Cloud Systems Lifecycle . . . 16
2.3 Complexity in Cloud Systems . . . 17
2.3.1 Cross-Component Interactions . . . 17
2.3.2 Dynamic Operating Conditions . . . 18
2.3.2.1 Change in Hardware conditions . . . 18
2.3.2.2 Change in Workload . . . 19
2.4 Overarching Challenges . . . 19

II Rapid Offline Prototyping with Blueprint 23

3 Characterizing Microservice Design Space and Usage 24
3.1 Microservice Design Space . . . 25
3.2 Microservices Usage and Modifications . . . 28
3.2.1 A Survey of Usage . . . 28
3.2.2 A Survey of Modifications . . . 32
3.3 Need for Modifying Microservices . . . 34
3.3.1 Use Case 1 (UC1): Design Space Exploration . . . 34
3.3.2 Use Case 2 (UC2): Reproducing Emergent Phenomena . 35
3.3.3 Use Case 3 (UC3): Prototyping and Evaluating Improvements . . . 37
3.3.4 Need for Configure, Build, Design (CBD) . . . 38
3.4 Manual Effort in Modifying Microservices . . . 39

4 Blueprint Design and Implementation 40
4.1 Approach . . . 42
4.1.1 Programmatically Specifying the Creative Component . 42
4.1.2 Automatically Executing the Mechanical Component . . 43
4.2 Anatomy of a Blueprint Application . . . 44
4.2.1 Workflow Spec . . . 44
4.2.2 Wiring Spec . . . 46

4.3 Intermediate Representation (IR) . . . 48
4.3.1 Types of IR nodes . . . 48
4.3.2 Encoding Dependencies . . . 51
4.3.3 Compiler Plugins . . . 52
4.4 Compilation . . . 54
4.4.1 Wiring & Workflow Spec to IR . . . 56
4.4.2 Custom Compiler Passes . . . 57
4.4.3 IR to Implementation . . . 58
4.4.3.1 Resolving Dependencies . . . 59
4.4.3.2 Plugin-Specific Artifact Generation . . . 60
4.4.3.3 Runtime Namespaces . . . 65
4.5 Executing Blueprint Applications . . . 67
4.6 Prototype . . . 68

5 Evaluation **69**
5.1 Experimental Setup . . . 69
5.2 UC1: Design Space Exploration . . . 70
5.3 UC2: Eliciting Emergent Phenomena . . . 72
5.3.1 Case Study I: Metastability Failures . . . 72
5.3.2 Case Study II: Cross-System Inconsistency . . . 75
5.4 UC3: Prototyping Improvements . . . 76
5.5 Generating Realistic Systems . . . 79
5.6 Extensibility of Blueprint . . . 82
5.7 Cost of Blueprint . . . 84

6 Related Work **86**
6.1 Tools for Microservice Generation . . . 86
6.2 Open-Source Microservice Benchmarks . . . 88
6.3 Configurability & Reusability . . . 89

III Online Specialization of Systems with Iridescent 90

7 Specializing Cloud Systems 91
7.1 Specialization is Effective but Complex . . . 91
7.1.1 A Case for Code Specialization . . . 93
7.1.2 Illustrative Example . . . 94
7.2 Specialization is Challenging in Practice . . . 95

8 Iridescent Design and Implementation 98
8.1 Approach . . . 99
8.1.1 Programmatically Specifying the Creative Component . 100
8.1.2 Automatically Executing the Mechanical Component . . 101
8.2 Anatomy of an Iridescent System . . . 102
8.3 Defining the Specialization Space . . . 106
8.3.1 Base Specialization Points . . . 106
8.3.2 Custom User-Defined Specialization Points . . . 109
8.3.3 Combining Specialization Points . . . 110
8.3.4 Specialized Optimization Pipelines . . . 111
8.3.5 Tracking Specialization State . . . 112
8.4 Defining Specialization Policies . . . 113
8.4.1 Exploring the Specialization Space . . . 114
8.4.2 End-to-End Metrics as Optimization Targets . . . 115
8.5 Specialization Runtime . . . 117
8.5.1 Specializer . . . 117
8.5.2 Online Compilation with JIT . . . 118
8.5.3 Correctness through Specialization Guards . . . 120
8.6 Prototype Implementation . . . 121
8.6.1 Specializer Implementation . . . 122

9 Evaluation 124
9.1 Experimental Setup . . . 125

9.1.1 Case Studies . . . 125
9.1.1.1 Tiling Benchmarks . . . 125
9.1.1.2 LibLPM . . . 125
9.1.1.3 TAS . . . 127
9.1.1.4 FastClick . . . 128
9.1.1.5 Network Functions . . . 129
9.1.2 Experimental Testbed . . . 129
9.2 Compiler Optimizations at Runtime . . . 129
9.3 Runtime Design Exploration . . . 134
9.4 Overhead of Iridescent's Abstractions . . . 140
9.5 Summary of Results . . . 144

10 Related Work **146**
10.1 Offline Automatic Specialization . . . 146
10.2 Online Automatic Optimizations . . . 147

11 Combining Offline Prototyping with Online Specialization **151**
11.1 CBDS: Configure, Build, Deploy, Specialize . . . 152
11.1.1 Specializations in Workflow . . . 153
11.1.2 Specializations in Scaffolding . . . 154
11.2 Raglan: Iridescent as a Blueprint Plugin . . . 155
11.2.1 Raglan Specialization Runtime . . . 155
11.2.1.1 Emulating a JIT for Go . . . 155
11.2.1.2 Custom Specialization Passes . . . 157
11.2.2 Raglan Implementation . . . 159
11.2.3 Future Extensions . . . 160
11.3 Case Study & Evaluation . . . 161
11.3.1 Geo-Sharded User Service . . . 161
11.3.2 Experimental Results . . . 164
11.3.2.1 Experimental Setup . . . 164
11.3.2.2 Plugin Complexity . . . 165

11.3.2.3 Quantitative Results . . . 166
11.3.2.4 Results Summary . . . 170
11.4 Summary . . . 170

IV Conclusion 172

12 Conclusion 173
12.1 Summary of Results . . . 173
12.2 Future Work . . . 175
12.2.1 HITLSDA for Problem Diagnosis and System Evolution 175
12.2.2 Representative Large-Scale Microservice Testbeds . . . 177
12.2.3 Enabling Further Automation with Agentic AI . . . 179

V Appendix 220

A Plugins for Executing Blueprint Applications 221
A.1 Blueprint's Testing Plugin . . . 221
A.1.1 Go Testing Framework . . . 223
A.1.2 Workflow-based Testing . . . 223
A.2 Blueprint's Workload Plugin . . . 225
A.2.1 Plugin APIs . . . 226
A.2.2 Plugin Usage . . . 228

List of Figures

1.1 Human-in-the-loop Specification-Driven Automation (HITLSDA) 5

2.1 Structure and Lifecycle of modern cloud systems 16

3.1 Excerpt from the ComposePostService implementation in the social network application from the DeathStarBench suite [139]. The code shown implements the ComposeCreator call to the UserService as part of the ComposePost API exposed by the ComposePost service. Code shown in purple corresponds to the application workflow, in orange to the scaffolding, and in pink to the instantiations. 26

4.1 Workflow Spec for DSB Social Network ComposePost. Lines 15-18 implement the functionality shown in Fig. 3.1 45

4.2 Built-in interface for a *cache* backend component. 45

4.3 Wiring Spec for three dependent DSB services. Zipkin tracing is enabled for all services; they are deployed in Docker containers; and communicate using gRPC. Cache and database instantiations are Memcached and MongoDB respectively. The color coding matches the color coding shown in Fig. 3.1 45
4.4 A depiction of Blueprint's IR for the three DSB services outlined in Fig. 4.3. Node shape is based on node type, and nodes are colored according to their granularity. Edges are color-coded according to the type of relationship. 48
4.6 Code excerpt from the compilation procedure of Blueprint . . . 55
4.7 Blueprint support for custom passes over the Blueprint IR . . . 57
4.8 Generated code for adding opentelemetry-compatible tracing to the ComposePost Service from the workflow 61
4.9 Generated code for running ComposePostService as a GRPC server. (Request unmarshall and Response marshall functions omitted for brevity) 62
4.10 Generated Code for Compose Post Process Node that uses Blueprint's runtime namespace for building instances of generated object definitions 64
4.11 Generated Code excerpt for a runtime namespace 66

5.2 Four metastability failure types [186] demonstrated in the BP-DSB-HR and BP-DSB-SN. 73

5.3 Metastability Vulnerability Analysis for BP-DSB-HR. 74
5.4 Cross-system inconsistencies arise in BP-DSB-SN when enabling replication. 75
5.5 Blueprint's reproduction of Fig. 6 from the Sifter paper [234] in SocialNetwork. 77
5.6 Prototype Solution for Type 1 Metastability failure in HotelReservation. 78
5.7 Performance comparison of Blueprint systems with their original counterparts 79
5.12 DSB-SN cache choice exploration 84

8.2 MMulBlockBench Microbenchmark 103
8.4 Architecture of an Iridescent-specialized system. 105
8.5 Demonstration of the Base Specialization Points 107
8.6 Interface for implementing custom code generation specialization points 109
8.7 Specialization Context that stores the specialization state of the system 112
8.8 Interface for implementing custom exploration strategies 114
8.9 Iridescent support for supporting end-to-end metrics as optimization targets 116
8.10 Contents in JIT for a specialized function 119

9.1 Sample LibLPM-NI generated code 126

9.3 Average cycles required to perform a table lookup with varying table sizes under two different workloads. 130
9.4 FastPath router throughput with Iridescent specialized LinearIPLookup element for different routing table sizes at 100% fast-path hit rate. 132
9.5 FastPath router throughput with Iridescent specialized LinearIPLookup element for varying fast-path hit rates. 132
9.6 Automatic exploration and specialization of different configurations for TAS. 135
9.7 Automatic exploration and specialization of MMulBlockBench for two different workloads. 136
9.8 Batch size exploration with Network Functions 137
9.9 Optimal fast path size exploration with Iridescent. 138
9.11 Compilation time of LibLPM-NI (nested-if specialization) as a function of the number of elements in the LPM lookup table. . . 141
9.13 Impact of sampling rate on instrumentation overhead for LibLPM-FP configuration for Workload B. 142

11.1 Fixed Order implementation of `GetUserInfo` as generated by Blueprint 161
11.2 Online Adaptation for the UserService case study with Raglan in comparison to other baseline configurations 167
11.3 Raglan exploration comparison with Smith's Rule with minimizing 90th percentile latency as the objective function 169

A.1 Sample End-to-End Blackbox test for UserService from the Social Network application 222
A.2 APIs exported by Blueprint's `workload` plugin 226
A.3 Sample usage of the workload plugin for the Social Network application in its workflow spec 227

List of Tables

3.2 Microservice systems used in the literature. 29

3.3 Modified Forks for each Microservice Benchmark 32

4.5 Types of IR nodes in Blueprint. 49

5.1 Lines of code of original and Blueprint implementations of popular open-source microservice systems 72

5.8 Lines of Code required for adding a backend interface. 81

5.9 Lines of Code required for adding a new instantiation. 81

5.10 Lines of code required for adding a new compiler plugin. 83

5.11 Time taken by Blueprint for generating the system including services, caches, databases, queues, tracers 83

7.1 Optimal value of block size, B, for our block matrix multiply, across 5 hardware platforms and 3 workloads. 95

8.1 Iridescent specialization types and their effects. 99

8.3 Iridescent API overview. 104

9.2 Impact of turning block size, B, as compile-time constant at run-time for N=64 for different hardwares (Tab. 7.1) 130
9.10 JIT compilation time and corresponding size 141
9.12 Lines of Code change required to integrate Iridescent. 142

List of Abbreviations

Abbreviation	Full Form
CBD	Configure, Build, Deploy
CBDS	Configure, Build, Deploy, Specialize
BP-DSB-SN	Blueprint's DeathStarBench's SocialNetwork
BP-DSB-HR	Blueprint's DeathStarBench's HotelReservation
HITLSDA	Human-in-the-loop Specification-Driven Automation
UC	Use Case

Part I

Introduction

Chapter 1

Introduction

Cloud systems form the backbone of today's computing landscape. These systems power the vast majority of the applications we use, including web search, social networks, e-commerce websites, reservation systems, banking applications, ride-sharing services, instant delivery applications, and many others.

Heterogeneous Architectures. Cloud applications are typically developed as suites of microservices comprising a large number of small, independent, loosely coupled services, each providing a unique capability. Typically, each service is independently developed, optimized, deployed, and orchestrated. To implement and orchestrate their business functionality, services can choose from a wide variety of frameworks and libraries that provide the necessary features and orchestration capabilities. Services typically communicate with one another over the network to compose workflows that execute the system's full functionality.

Shared, Dynamic Deployments. These applications run on shared virtualized

hardware in datacenters. The services comprising the system may be distributed across different machines within a single physical datacenter or geo-distributed across multiple datacenters around the world. To maximize resource efficiency, cloud infrastructure providers often avoid allocating an entire machine to each service and instead co-locate multiple services on a single machine. Moreover, due to the dynamic nature of workloads, services may be moved from one machine to another at runtime or independently scaled across multiple machines.

The heterogeneous and dynamic nature of cloud systems introduces significant complexity during the cloud lifecycle. First, designing and implementing complex cloud systems is difficult because the design space is vast, and it is challenging to predict *a priori* which design is best suited for a given system. Moreover, predicting the performance implications of interactions between components before executing the system is nearly impossible, making it difficult to select an optimal design. Second, extracting the best performance from systems at all times is challenging because the operating conditions under which cloud systems are deployed often change at runtime. This commonly occurs for two reasons: (i) changes in hardware due to migration and (ii) changes in workload. Thus, even if developers design an optimal system, it may not remain optimal under all possible operating conditions. Third, the complexity of cloud systems gives rise to unpredictable problematic end-to-end behaviors rising from interactions between the different components. Consequently, timely incident detection and resolution require significant manual effort [217, 282] as it is challenging to identify all metrics a system might need prior to deployment [141, 150]. A lack of

relevant signals and monitors can lead to misses in incident detection [141, 150], while ad-hoc metrics can generate noise leading to false positives [141]. Finally, developers must continuously improve systems to accommodate evolving functional requirements. However, this process is tedious because developers must ensure that any changes introduced while adding new functionality preserve the correctness of existing functionality, requiring significant engineering effort to safely evolve the system.

The core issue across the cloud system lifecycle is that developers must expend significant manual effort to perform these tasks due to the complexity of the systems. Consequently, there is a need to develop novel abstractions and tooling that fundamentally rethink how developers design, implement, and optimize systems today, with a focus on fully or partially automating tasks that currently require developers to manually manage this complexity.

1.1 Thesis Statement and Goal

Thesis Statement. Cloud systems are large, diverse, and complex. Designing, optimizing, operating, and evolving these systems requires a significant amount of manual effort. Human-in-the-loop Specification-Driven Automation (HITLSDA) techniques for the design exploration and optimization of cloud systems address this challenge by enabling developers to better manage system complexity and reduce the manual effort required to design, implement, and optimize cloud systems.

Thesis Goal. The goal of this thesis is to investigate longstanding challenges

in managing the complexity of cloud systems. To address these challenges, this thesis proposes new abstractions that reduce the manual effort required from developers to automate the design, implementation, and optimization of cloud systems.

1.2 Thesis Contributions

1.2.1 Human-in-the-loop Specification-Driven Automation

This thesis tackles the central problem of the substantial amount of manual and mechanical effort required to design efficient and reliable cloud systems. To reduce the manual and mechanical burden on users, my research focuses on developing HITLSDA techniques to enable users to focus on system design tasks requiring human insight and creativity.

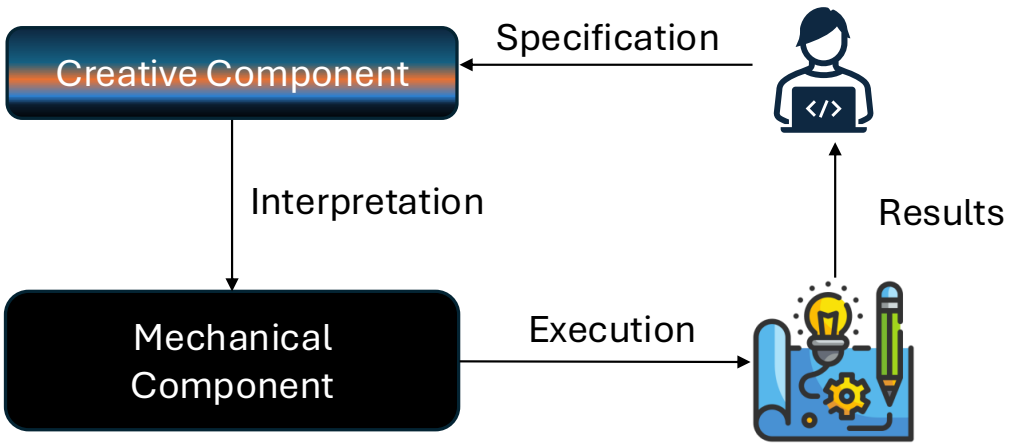


Fig. 1.1: Human-in-the-loop Specification-Driven Automation (HITLSDA)

Breaking down the Manual Effort. The key insight in my thesis is that cloud tasks can be conceptually decomposed into two components — (i) a creative component that requires human insight; and (ii) a mechanical component to interpret and execute the creative component. For example, consider the task of selecting a cloud system design. The creative component that requires human insight is selecting potential designs that are likely candidates that fit the desired requirements. The mechanical component is converting these possible designs into implementations and subsequently benchmarking them to find the best design. The large manual effort required by users is due to the tedious and manual nature of the mechanical component.

My research leverages this insight by developing abstractions and building corresponding automation tools to natively support HITLSDA as shown in Fig. 1.1. HITLSDA is based on the following two key ideas.

Programmatically Specifying the Creative Component. The creative component of cloud tasks can be programmatically specified. To do so, I design abstractions tailored for users to specify their creative component. To support easy design space exploration of microservice systems, I have developed new specification formats as part of the Blueprint project (Part II) to specify the different design dimensions of the system.

Automatically Executing the Mechanical Component. A large fraction of the human effort required for the cloud lifecycle is the mechanical effort. The key idea is that with the right tools, this mechanical effort can be fully automated. To do so, as part of my research, I design automation tools that take as input

the programmatically specified creative component and convert that into concrete executions. For example, to generate implementations of microservice systems from the programmatically specified design choices as part of the Blueprint project (Part II), I have developed a novel compiler that automatically generates the implementation of the microservice system that satisfies the specified design.

1.2.2 Two HITLSDA Instances for the Cloud Lifecycle

This thesis introduces two novel HITLSDA tools to reduce the manual cost required to build efficient modular cloud systems. Specifically, this thesis focuses on reducing the manual effort involved in offline design exploration of modular cloud systems and in dynamically specializing systems to achieve optimized performance across different deployment settings.

1.2.2.1 Rapid Offline Design Prototyping with Blueprint

Modern microservice development often requires repeatedly exploring alternative system designs, configurations, and deployment strategies, yet much of this process remains manual and labor-intensive. To address this challenge, I have developed Blueprint (Chapter 4), a toolchain for rapidly generating and reconfiguring microservice systems. Blueprint enables developers to programmatically specify high-level design choices while automating the mechanical process of constructing a complete deployable implementation. As a HITLSDA technique, Blueprint separates the creative component of system design from the repetitive engineering effort required to realize that design in practice by allowing users to

programmatically specify their desired design through Blueprint's input specifications which Blueprint's compiler automatically converts into a complete, runnable implementation. Blueprint's abstractions and compiler significantly accelerates offline design-space exploration by reducing the amount of handwritten code required to implement, modify, build, and deploy microservice variants, thereby making iterative experimentation substantially easier.

1.2.2.2 Online Code Specialization of Systems with Iridescent

Modern systems operate under continuously changing workloads and environments, making static optimization insufficient for achieving sustained high performance. To address this challenge, I have developed Iridescent (Chapter 8), a framework for online, workload-driven runtime code specialization that dynamically adapts system implementations based on observed execution behavior. Iridescent provides a toolkit that allows developers to define system-specific specialization opportunities by exposing runtime data and invariants that can be leveraged to generate more efficient execution paths. As a HITLSDA technique, it separates the creative component—the specification of the specialization space—from the mechanical component of exploring and selecting among possible specialized implementations. The Iridescent runtime then systematically navigates this space according to a user-provided policy to identify high-performing combinations of specializations under current workload conditions. By combining developer-provided structure with automated compiler-driven optimization, Iridescent enables systems to continuously adapt to shifting workloads and execution environments while minimizing manual tuning effort.

1.3 Publications and Collaborations

1.3.1 Thesis Publications

This thesis contains text and material from the following two publications [14, 15]:

- "Blueprint: A toolchain for highly-reconfigurable microservice applications." Vaastav Anand, Deepak Garg, Antoine Kaufmann, Jonathan Mace. *29th Symposium on Operating Systems Principles, 2023.*
- "Iridescent: A framework enabling online system implementation specialization". Vaastav Anand, Deepak Garg, Antoine Kaufmann. *Under submission. arXiv preprint available at* `https://arxiv.org/pdf/2508.16690`.

The following paragraphs describe the origins of the text and material in this thesis in more detail.

Chapter 1. The introduction is entirely written by me.

Chapter 2. The cloud systems structure and lifecycle chapter is entirely written by me. The text of §2.1 derives text from the Blueprint paper [15], which was written by me in collaboration with my advisors.

Chapter 3. The text of Chapter 3 is derived from the Blueprint paper [15] and its supplementary material, which was written by me in collaboration with my advisors.

Chapter 4. The text of Chapter 4 is derived from the Blueprint paper [15] and its supplementary material, which was written by me in collaboration with my advi-

sors. The text has been extended with details for better clarity. The code snippets have been updated to reflect the current open-source version of Blueprint.

Chapter 5. The text of Chapter 5 is derived from the Blueprint paper [15] and its supplementary material, which was written by me in collaboration with my advisors. The lines-of-code numbers in §5.7 have been updated to use the numbers for the current open-source version of Blueprint.

Chapter 6. The text of Chapter 5 is derived from the Blueprint paper [15], which was written by me in collaboration with my advisors.

Chapter 7. The text of Chapter 7 is derived from the Iridescent paper [14], which was written by me in collaboration with my advisors.

Chapter 8. The text of Chapter 8 is derived from the Iridescent paper [14], which was written by me in collaboration with my advisors. The text has been extended with details for better clarity.

Chapter 9. The text of Chapter 9 is derived from the Iridescent paper [14], which was written by me in collaboration with my advisors. The text has been extended with details for better clarity.

Chapter 10. The text of Chapter 10 is derived from the Iridescent paper [14], which was written by me in collaboration with my advisors. The text has been extended with details for better clarity.

Chapter 11. The text of Chapter 11 is entirely written by me. The chapter provides insights into how to effectively combine rapid offline prototyping (Blueprint) with online code specializations (Iridescent). The chapter also introduces an

implementation of Iridescent's abstractions that are compatible with Blueprint specifications.

Chapter 12. The text of the Chapter 12 is entirely written by me. The section on automating cloud tasks with Agentic AI (§12.2.3) is a summary of an ongoing collaboration with the following researchers at Microsoft: Rodrigo Fonseca, Ricardo Bianchini, Jonathan Mace, Pedro Las-Casas, Alok Gautam Kumbhare, Gagan Somashekar, Chetan Bansal, Celine Irvene as well as a former intern at Microsoft, Yichen Li. The summary is derived from three workshop papers, "Intent-based System Design and Operation" [18], "Automated Service Design with Cerulean (Project Showcase)" [16], and "Towards Using LLMs for Distributed Trace Comparison (Abstract)" [17].

1.3.2 Non-Thesis Publications

During my PhD, I have also contributed to the following publications [16–18, 21, 23, 88, 101, 153, 260, 270, 365, 456], which are not contributions of this thesis.

- "The Odd One Out: Energy is not like Other Metrics".
  Vaastav Anand, Zhiqiang Xie, Matheus Stolet, Roberta De Viti, Thomas Davidson, Reyhaneh Karimipour, Jonathan Mace. *1st HotCarbon: Workshop on Sustainable Computer Systems, 2022.*
- "The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems".
  Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, Jonathan Mace. *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2023.*
- "Towards Using LLMs for Distributed Trace Comparison (Abstract)".
  Vaastav Anand, Pedro Las-Casas, Rodrigo Fonseca, Antoine Kaufmann. *6th*

International Workshop on Cloud Intelligence/AIOps Workshop (co-located with ICSE), 2025.

- "Automated Service Design with Cerulean (Project Showcase)".
  Vaastav Anand, Alok Gautam Kumbhare, Celine Irvene, Chetan Bansal, Gagan Somashekar, Jonathan Mace, Pedro Las-Casas, Ricardo Bianchini, Rodrigo Fonseca. *6th International Workshop on Cloud Intelligence/AIOps Workshop (co-located with ICSE), 2025.*

- "Intent-based System Design and Operation."
  Vaastav Anand, Yichen Li, Alok Gautam Kumbhare, Celine Irvene, Chetan Bansal, Gagan Somashekar, Jonathan Mace, Pedro Las-Casas, Ricardo Bianchini, Rodrigo Fonseca. *4th Workshop on Practical Adoption Challenges of ML for Systems (PACMI'25) (co-located with SOSP), 2025.*

- "Generating representative macrobenchmark microservice systems from distributed traces with Palette."
  Vaastav Anand, Matheus Stolet, Jonathan Mace, Antoine Kaufmann. *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2025).*

- "DMAS-Forge: A Framework for Transparent Deployment of AI Applications as Distributed Systems".
  Alessandro Cornacchia, Vaastav Anand, Muhammad Bilal, Zaffar Ayub Qazi, Marco Canini. *1st Workshop on Systems for Agentic AI (SAA'25) (co-located with SOSP).*

- "MAESTRO: Multi-Agent Evaluation Suite for Testing, Reliability, and Observability."
  Tie Ma, Yixi Chen, Vaastav Anand, Alessandro Cornacchia, Amandio Faustino, Guanheng Liu, Shan Zhang, Hongbin Luo, Suhaib Fahmy, Zafar Qazi, Marco Canini. *Under submission. Pre-print available on arXiv at* https://arxiv.org/pdf/2601.00481.

- "SoK: The Faults in our Graph Benchmarks".

Puneet Mehrotra, Vaastav Anand, Daniel Margo, Milad Hajdehi, Margo Seltzer. *Pre-print available on arXiv at* https://arxiv.org/pdf/2404.00766.

- "Columbo: Low Level End-to-End System Traces through Modular Full-System Simulation."
  Jakob Görgen, Vaastav Anand, Hejing Li, Jialin Li, Antoine Kaufmann. *Pre-print available on arXiv at* https://arxiv.org/pdf/2408.05251.
- "Chatting with Logs: An Exploratory Study on Finetuning LLMs for LogQL".
  Vishwanath Seshagiri, Siddharth Balyan, Vaastav Anand, Kaustubh Dhole, Ishan Sharma, Avani Wildani, José Camrbonero, Andreas Zülfe. *Pre-print available on arXiv at* https://arxiv.org/pdf/2412.03612.
- "SoK: Demystifying the multiverse of MPC protocols."
  Roberta De Viti, Vaastav Anand, Pierfrancesco Ingo, Deepak Garg. *Pre-print available on arXiv at* https://arxiv.org/pdf/2512.11699.

Chapter 2

A Primer on Modular Cloud Systems

Modern cloud systems are typically deployed on public cloud platforms, run continuously, and require ongoing development, maintenance, and optimization. In this background section, we briefly introduce the typical structure of cloud systems, describe the key tasks developers perform throughout a typical cloud system lifecycle, and explain the various sources and implications of system complexity.

2.1 Cloud Systems Structure

Cloud systems are typically structured as distributed, service-oriented platforms composed of many loosely coupled components that communicate over networks. Instead of building large monolithic applications, cloud providers and

cloud-native applications are increasingly designed around modular services, each operating independently while integrating through well-defined APIs. To achieve modularity, these applications are often developed as suites of loosely coupled microservices [85]. Consequently, such applications tend to become large, diverse, and complex while still preserving modularity. For example, Uber reported hosting over 6,000 microservices in 2025 [237], while Meta reported over 18,500 active services and more than 12 million instances in 2023 [188].

Advantages of Microservices. The modular structure of microservice-based applications allows developers to design, test, and deploy each service independently. This improves the development speed of cloud systems, as changes to one service can be tested without directly impacting other parts of the overall system. Because services are independent, failures in one service may not directly affect the rest of the system, thereby improving maintainability and reducing the risk of system-wide failures. Moreover, this modular approach enhances scalability, as individual components can be scaled independently based on workload rather than scaling the entire application, resulting in more efficient resource utilization.

Disadvantages of Microservices. While modularity improves maintainability, scalability, and development speed, it also significantly increases system complexity, as developers must now manage the intricate interactions between different services as well as their supporting components. Moreover, debugging and monitoring these applications becomes more difficult, as errors must be traced across interactions among multiple services.

2.2 Cloud Systems Lifecycle

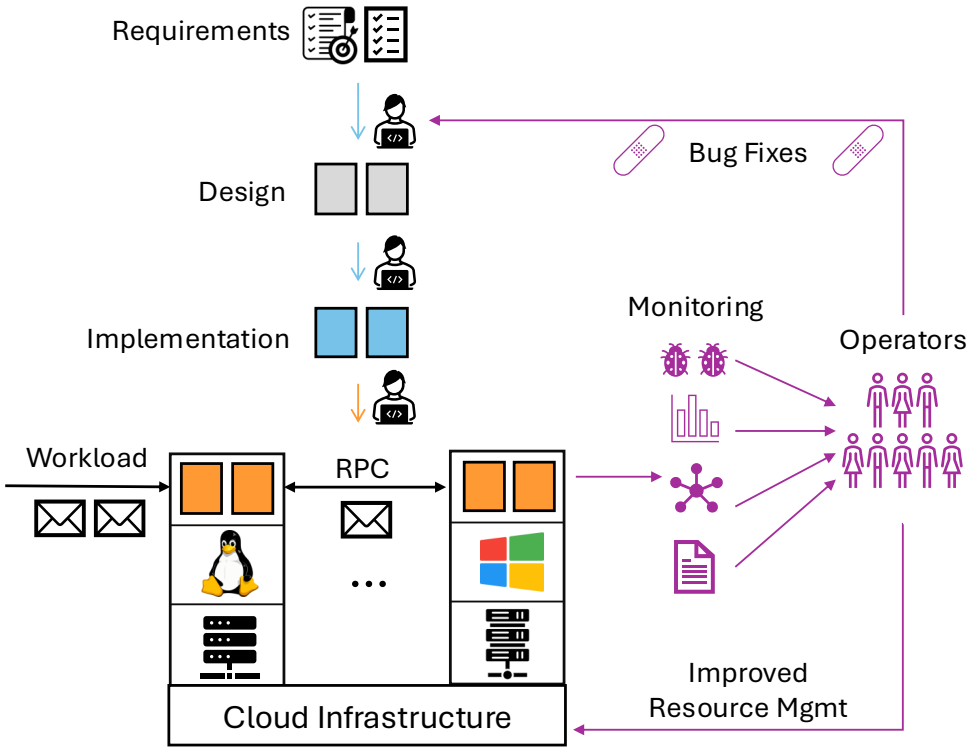


Fig. 2.1: Structure and Lifecycle of modern cloud systems

Fig. 2.1 shows the different phases of the lifecycle of modern cloud systems. These phases include an initial design and implementation phase, in which the requirements of various stakeholders are translated into an implementable design; an optimization phase, in which the selected design is tuned to handle typical workloads; an operation phase, in which the system is monitored during execution to detect performance regressions or bugs; and a system improvement

phase, in which developers triage and analyze the root causes of bugs and performance regressions to determine how the system should be improved. These phases are repeated throughout the entire lifetime of the cloud system.

2.3 Complexity in Cloud Systems

Complexity detrimentally impacts the cloud lifecycle by making it significantly harder for developers to efficiently carry out cloud lifecycle tasks. In this chapter, we discuss two main sources of complexity in modern cloud systems.

2.3.1 Cross-Component Interactions

Cloud systems are composed of multiple independent subsystems that are loosely-connected through interactions. These interactions can be synchronous or asynchronous and may take various forms, including direct RPC calls, indirect queues, resource allocations, filesystem reads/writes, among others. As these systems are often developed by independent teams of developers, teams can often make conflicting or incorrect assumptions about the interactions leading to bugs and failures. A recent study found that cross-system interaction failures account for around 20% of the most catastrophic cloud incidents [400].

Emergent Misbehaviors. Emergent phenomena, or emergent behaviors [277], are a special class of cross-component interactions that manifest as aspects of a system;s runtime behavior not localized to any single service or component. Instead, they arise as the cumulative effect of interactions *between* components under a given workload and application design. Emergent phenomena encom-

pass end-to-end performance, correctness, and reliability concerns of an application. Notable examples include cascading failures [310, 414, 459], tail latency effects [102], cross-system consistency issues [5], laser of death [294], and metastable failures [57, 186]. Left unchecked, emergent phenomena can lead to degraded service and even outages [268, 290, 294]; thus they are the focus of a range of recent work in both industry [372] and academia [57, 186, 254, 322].

These issues are challenging to triage and even more difficult to predict, as they arise primarily from complex interactions under specific, often atypical, conditions. Designing systems that are robust to such issues remains a critical challenge for developers of modern cloud systems.

2.3.2 Dynamic Operating Conditions

The conditions under which cloud systems operate are dynamic and often change. This dynamism typically manifests in two common ways: changes in deployment hardware and changes in workload. We describe them in detail below.

2.3.2.1 Change in Hardware conditions

Cloud systems are deployed on a shared substrate of heterogeneous hardware resources operated by cloud providers. This substrate must be shared among all tenants, with the cloud provider responsible for managing system placement, *i.e.* determining on which hardware each system executes.

To improve hardware utilization, cloud providers may migrate systems to different machines at runtime. This can result in a significant change in the operating environment by completely altering the underlying hardware on which

a system was previously executing.

Cloud providers also co-locate multiple systems on the same hardware, leading to multiple applications competing for shared resources. As a result, the performance of one application or tenant can be significantly impacted by co-located workloads, either because a co-tenant consumes a disproportionate share of resources [35, 261] or simply due to interference effects, *e.g.* increased cache misses or other forms of resource contention.

2.3.2.2 Change in Workload

Cloud systems are designed to handle a wide variety of workloads. Workloads exhibit periodicity, with fluctuations occurring over short time scales. These fluctuations can cause systems to behave suboptimally, as a selected design may only be optimal for a subset of the workload. Changes in workload can therefore degrade both the performance and efficiency of these systems. In the worst case, a drastic change in workload can cause the system to "fall off the cliff" and exhibit non-linear behavior [83], potentially leading to partial or even complete system collapse.

As these operating conditions are not known *a priori* and do not remain static throughout system execution, ensuring the efficiency of cloud systems over their entire lifetime remains a critical challenge today.

2.4 Overarching Challenges

The sources of complexity make system design, optimization, and evolution difficult. Specifically, users face four particularly challenging tasks:

- **Selecting the best design.** For any microservice application, there are many possible designs, each with its own performance characteristics and potential issues. Predicting the performance of a design prior to executing the system is nearly impossible due to complex interactions between components. Consequently, users must manually explore candidate designs by converting each into a concrete implementation and measuring its performance. This requires significant manual effort, as users must build multiple implementations of the system to identify a viable design, which is both expensive and time-consuming.

- **Optimizing for best performance.** Cloud systems operate under dynamic conditions: workloads change frequently, infrastructure is shared with other applications, components fail unpredictably, and cloud providers may migrate applications across different hardware. As a result, a design that is optimal under one condition may not remain optimal across all operating conditions. Typically, developers must choose at implementation time between specializing the system for a specific workload or adopting a more general design that is less performant but applicable to a broader range of workloads. This decision is further complicated by the fact that optimal implementation choices often depend on runtime information that is not available during offline optimization. As a result, specialization decisions are ideally made at runtime when full information is available; however, this is often infeasible, as it would require developers to be continuously available to make significant live system changes.

- **Understanding problematic behaviors.** At scale, unpredictable behaviors are common, such as metastability failures and anomalies that emerge from complex cross-component interactions. These misbehaviors are difficult to predict *a priori* and even more difficult to test for during the design phase. Moreover, when such issues occur at runtime, root-cause analysis is labor-intensive, requiring developers to sift through large volumes of data from multiple sources to identify relevant signals and diagnose the issue. This is further complicated by the fact that observability data may be incomplete, as instrumentation is often limited for performance reasons, potentially omitting critical information needed to identify the true root cause. Moreover, when these misbehaviors occur at runtime, analyzing these behaviors to find the root causes is especially laborious as developers must sift through large swathes of data from multiple sources to identify relevant data and then further analyze data to triage issues. This is further complicated by the fact that the data might be missing the true root cause of the issue because of a lack of detail in the instrumentation for generating observability data due to performance reasons.

- **Evolving and improving systems.** The functional and operational requirements of a system change over time. These changes may affect different abstraction levels of the system (*e.g.* architectural, interface, or implementation) and require significant effort, as developers must first identify the precise modifications needed to support new requirements and then implement them correctly without breaking existing functionality. Currently, such changes are applied manually by localizing modifications to specific parts of the codebase

that implement the required functionality.

In this thesis, we present HITLSDA techniques for overcoming the challenges in the first two tasks. We leave building HITLSDA techniques for the remaining two tasks as future work.

Part II

Rapid Offline Prototyping with Blueprint

Chapter 3

Characterizing Microservice Design Space and Usage

Microservice architecture is the widely used approach for building modular cloud systems. The microservice architecture approach decomposes applications into smaller, modular services that communicate over the network. As a result of their success in enabling large-scale and continually-evolving applications, microservices have become ubiquitous among internet companies [84, 85, 171, 173].

In this section, we characterize the design space of microservice applications, analyze and understand how these applications are developed and modified, and explain the key reason as to why building and modifying microservice systems requires significant manual effort.

3.1 Microservice Design Space

The space of possible designs for microservice applications is enormous, and while there may be guiding high-level design principles, every application differs substantially from the next. In this section, we briefly elaborate three important dimensions for the design of a microservice application – the ***application-level workflow***; the lower-level ***scaffolding*** infrastructure on which the workflow executes; and ***instantiations*** for different infrastructure implementations.

Application-Level Workflow. The application-level workflow of a microservice system refers to the core application-level business logic and end-to-end flow of executions through the system's different components. Microservice applications vary widely in terms of their application-level workflow. For example, recent open-source microservice benchmarks cover diverse domains, such as e-commerce apps [155, 273, 418, 425], social networks [139], and reservation systems [139, 468] among others [3, 29, 139, 195, 393]. Additionally, these applications differ in the number of services and APIs they use internally from only a few (SockShop [273]) to dozens (TrainTicket [468]). Even applications with similar high-level goals (*e.g.* TrainTicket [468] and DSB Hotel Reservation [139]) have vastly different workflows.

Scaffolding. Scaffolding refers to the design of the deployment and orchestration setup used to execute an application, and is orthogonal to (and typically opaque to) the application's core workflow. It defines *what kinds of* mechanisms are used to support execution, such as what type of communication frameworks

```
class ComposePostHandler :  public ComposePostThriftService {
    ClientPool<ThriftClient<UserServiceClient>> *_user_service_client_pool;
};

Creator ComposePostHandler::_ComposeCreaterHelper(int64_t req_id, int64_t user_id, const std::string
     &username,const std::map<std::string, std::string> &carrier) {
    TextMapReader reader(carrier);
    auto parent_span = opentracing::Tracer::Global()->Extract(reader);
    auto span = opentracing::Tracer::Global()->StartSpan("compose_creator_client",
         {opentracing::ChildOf(parent_span->get())});
    std::map<std::string, std::string> writer_text_map;
    TextMapWriter writer(writer_text_map);
    opentracing::Tracer::Global()->Inject(span->context(), writer);

    auto user_client_wrapper = _user_service_client_pool->Pop();
    if (!user_client_wrapper) {
        ServiceException se;
        se.errorCode = ErrorCode::SE_THRIFT_CONN_ERROR;
        se.message = "Failed to connect to user-service";
        LOG(error) << se.message;
        span->Finish();
        throw se;
    }
    auto user_client = user_client_wrapper->GetClient();
    Creator _return_creator;
    try {
        user_client->ComposeCreatorWithUserId(_return_creator, req_id, user_id,
             username,writer_text_map);
    } catch (...)  {
        LOG(error) << "Failed to send compose-creator to user-service";
        _user_service_client_pool->Remove(user_client_wrapper);
        span->Finish();
        throw;
    }
    _user_service_client_pool->Keepalive(user_client_wrapper);
    span->Finish();
    return _return_creator;
}
```

Fig. 3.1: Excerpt from the ComposePostService implementation in the social network application from the DeathStarBench suite [139]. The code shown implements the ComposeCreator call to the UserService as part of the ComposePost API exposed by the ComposePost service. Code shown in purple corresponds to the application workflow, in orange to the scaffolding, and in pink to the instantiations.

are used, whether and how services are replicated, and how requests are load balanced, among other concerns. These components orchestrate the execution of core functional behavior and augment it with additional non-functional capabilities such as horizontal scaling and observability. Scaffolding includes middleware, frameworks, libraries, and backend systems that provide features such as RPCs, replication, load balancing, circuit breakers, and more [36]. Importantly, scaffolding can be changed without modifying the application's workflow or functional behavior. For example, the original DSB Social Network [139] uses a single runtime instance per service by default; however, a modified version of the application could change its scaffolding to replicate service instances without altering the end-to-end application workflow [253].

Instantiations. For every piece of scaffolding there may be different implementations to choose from, configuration dimensions of those implementations, and choices for configuration parameters, *e.g.* to enable RPC an application may use gRPC, Thrift, or some research prototype framework [213].

Fig. 3.1 shows an excerpt from the implementation of the `ComposePostService` of the SocialNetwork application in DeathStarBench. Specifically, the excerpt shows the implementation of one outgoing call from the `ComposePostService` to the `UserService` for the `ComposeCreator` API. To show how the different dimensions of a microservice application manifest in real implementations, we have color-coded the excerpt with three different colours - purple shows the application workflow, orange shows the scaffolding, and pink shows the instantiations. The key takeaway is that all three dimensions are tightly coupled together in the

code. This tight coupling makes it non-trivial to tease apart and change the three dimensions independently.

3.2 Microservices Usage and Modifications

To better understand how users use and modify open-source microservice systems, we conducted two complementary studies. First, we performed a literature survey to analyze which systems are most commonly used in research and the different experimental scenarios for which they are employed. Second, we analyzed forks and modifications of popular benchmark systems to understand the types of changes developers and researchers make in practice and the engineering effort required to implement them. Together, these studies help identify the recurring use-cases, customization requirements, and modification patterns of microservice systems. In particular, our analysis shows that users frequently adapt systems by changing architectural structures, deployment configurations, communication mechanisms, observability infrastructure, workloads, and runtime behavior, often requiring substantial manual effort even for conceptually simple modifications.

3.2.1 A Survey of Usage

To understand how microservice systems are used, we conducted a literature survey to analyze which systems were most commonly used in research. To identify relevant papers, we began with a set of seed systems, namely DeathStarBench [139] and TrainTicket [468], and used Google Scholar to find papers

System	Count	Papers
DSB-SN [139]	51	[12, 20, 22, 37, 48, 49, 55, 56, 81, 93, 97, 123, 134, 135, 138, 140, 184, 187, 191, 200, 201, 205, 208, 212, 234, 242, 274, 300, 309, 310, 314, 318, 319, 322, 332, 333, 335, 356, 382, 383, 405, 424, 426, 453, 457, 458, 461, 462]
DSB-MM [139]	18	[49, 55, 56, 62, 95, 97, 134, 140, 212, 236, 274, 309, 322, 333, 335, 357, 382, 408]
DSB-HR [139]	17	[49, 81, 134, 140, 181–183, 207, 212, 227, 264, 322, 354, 371, 458, 470, 474]
DSB-Swarm [139]	1	[142]
μSuite [393]	2	[275, 394]
TrainTicket [468]	52	[2, 11, 28, 37, 60, 61, 64, 90, 94, 95, 98, 109, 134, 164, 169, 181–185, 187, 193, 205, 248, 249, 252, 255, 264, 265, 284, 322, 348, 349, 356, 357, 364, 382, 397, 409, 410, 420–422, 424, 452, 464–467]
SockShop [273]	96	[13, 24, 25, 31, 33, 46, 52, 54, 65, 66, 68, 71, 74, 79, 98, 100, 111, 115, 120, 124, 133, 157, 158, 163, 165–170, 181–183, 206, 209, 210, 212, 215, 216, 225, 226, 228–232, 243–246, 252, 262, 264, 265, 269, 271, 278, 283, 285–287, 289, 291, 298, 301, 323, 324, 330, 333, 340, 344, 346, 348, 352, 353, 364, 370, 379–381, 399, 401, 403, 406, 407, 411, 416, 423, 427, 432–434, 439–441, 445, 454, 466, 469]
AcmeAir [3]	26	[4, 40, 41, 51, 80, 108, 122, 159, 160, 190, 191, 206, 222, 281, 299, 337, 341, 363, 364, 398, 412, 413, 436, 443, 444]
TeaStore [418]	32	[32, 63, 93, 96, 117, 119–121, 129, 149, 154, 162–164, 172, 174, 180, 189, 194, 194, 221, 247, 266, 333, 355, 358, 359, 377, 378, 389, 417, 437]
OnlineBoutique [155]	34	[34, 37, 47, 67, 69, 72, 75, 78, 136, 137, 156, 176, 218, 227, 241, 245, 246, 252, 263, 271, 297, 310, 338, 348–351, 354, 381, 446–448, 451, 470]
AliBaba [257]	2	[258, 455]
Stan's RobotShop [425]	9	[42, 78, 208, 250, 286, 296–298, 310]
BookInfo [195]	12	[10, 30, 42, 78, 245, 246, 250, 250, 259, 292, 310, 348, 369]
eShopOnContainer [112]	13	[128, 143, 144, 197, 233, 265, 287, 295–298, 373, 385, 419]
LakeSideMutual [312]	10	[118, 143, 144, 240, 296, 326, 373, 374, 388, 471–473]
eShoppers [360]	3	[90, 91, 410]
PitStop [116]	2	[246, 293]
Lab Insurances Portal [29]	2	[208, 256]
Overleaf [303]	1	[208]
Retwis [336]	3	[142, 227, 361]

Tab. 3.2: Microservice systems used in the literature.

citing these systems. During analysis of the shortlisted papers, whenever we encountered a previously unseen system, we expanded our list of systems accordingly and added papers citing the newly identified system to the shortlist. Some systems did not have accompanying research papers, making it difficult to identify related work through citation analysis alone. To address this, we additionally queried Google Scholar using the system repository URLs to discover candidate papers using those systems.

TTo determine whether a paper used a particular system, we analyzed the paper's evaluation section. If the paper performed at least one experiment using a given system, we considered that paper to have used the system. In total, we identified 290 research papers collectively using 20 different systems. Of these, six systems—DSB-SN, DSB-HR, DSB-Swarm, DSB-MM, TrainTicket, and TeaStore—were explicitly described by their authors as microservice benchmarks. Eleven systems were demonstration applications accompanying specific frameworks or libraries to showcase particular features in a microservice context. Two systems were constructed from public trace datasets, and the final system, Overleaf [303], was the only publicly deployed production microservice system identified in the survey. While our survey identified 20 systems used in the literature, many additional open-source microservice systems exist [331].

Tab. 3.2 summarizes the papers identified in the survey grouped by the systems they used. Among all systems, SockShop [273] appeared in the largest number of papers despite consisting of relatively few services. We found it notable that SockShop was used more frequently than several systems explicitly

designed as microservice benchmarks. We believe this popularity is likely due to the system's longevity and the broad range of orchestration, deployment, and monitoring features it supports, which make it attractive for experimentation. This observation suggests that researchers value systems that expose a diverse set of operational and infrastructural features rather than merely large service counts.

While researchers use microservice systems to analyze and solve specific emergent phenomena, other use-cases of microservices include using them as sources of log and trace data, or as individual datapoints in a larger dataset of microservice systems for source code analysis. The use-cases of microservice systems are diverse and require a diverse set of systems that vary across a variety of dimensions. Thus, it is impossible that any single implementation of a microservice system can be used as the ideal benchmark system for all possible research scenarios.

Researchers use microservice systems for a wide variety of purposes beyond studying emergent runtime behavior. These systems are also used as sources of logs and traces, as datasets for source-code analysis, and as representative applications for evaluating deployment, observability, testing, scheduling, reliability, and performance techniques. Consequently, the requirements placed on benchmark systems are highly diverse and span many dimensions. No single microservice implementation can therefore serve as an ideal benchmark for all research scenarios.

3.2.2 A Survey of Modifications

Open-source system survey. To understand the modification effort required to change microservices, we analyzed forks of popular microservice systems to figure out what modifications were being made to the systems. We chose to anlayse the forks of the systems labeled as microservice benchmarks as these systems were designed as microservice benchmarks and are the ideal targets for modifications. We analyzed 240 forks of DeathStarBench [139], 130 forks of TrainTicket [468], and 96 forks of TeaStore [418]. Out of the total 464 forks, 146 forks made modifications to the benchmark systems. Tab. 3.3 shows the breakdown of the modified forks by system.

System	Total Forks	Forks w/ Modifications
DeathStarBench	240	85
TrainTicket	130	25
TeaStore	96	36

Tab. 3.3: Modified Forks for each Microservice Benchmark

We found multiple different types of modifications made to the systems. The most common modification to systems were to either enable or disable tracing and other monitoring tools. This included switching on/off tracing, metrics, and logging. Another common modification was to generate the manifest files for deploying systems on a specific framework such as Kubernetes or OpenShift. There were multiple forks that modified systems such that they exhibited a specific emergent phenomena. We found one fork of DeathStarBench that modified

the system to induce a metatstable failure [125, 126] into the system and another fork that modified the DeathStarBench system to include geo-replication to analyze cross-system inconsistencies [253]. Modifications were not restricted to emergent phenomena. Some other forks modified the systems to understand the impact of various design choices – for example, thrift vs grpc [43], energy efficiency of monitoring tools [110], among others. We also found a fork that modified TeaStore to study, reproduce, and exploit CVE-2021-44228 [339]. Researchers have also modified systems to support their experiments [19, 110, 438].

In pull request #101 of DSB SocialNetwork [106], the authors inverted caller-callee relationships between `ComposePostService` and `TextService`, `UserService`, `UniqueIDService`, and `MediaService` to improve application performance. They modified 5,140 LoC to achieve this change. We found that a popular change in existing microservice systems is to enable or disable tracing [19, 44, 178, 196]. For example, disabling tracing in a fork of DSB SocialNetwork required 418 LoC [196]. Authors of Sifter [234] manually modifies the DSB Social Network application to add X-Trace [131]. X-Trace is a distributed tracing framework, but it does not conform to OpenTelemetry APIs and cannot reuse the existing Jaeger instrumentation of DSB SocialNetwork. Consequently, the authors of Sifter manually extended DSB SocialNetwork to add X-Trace support. This comprised 1,289 LoC changed over a 2 week period, based on commit timestamps.

From this survey, we conclude that there exists a problem in making modifications to microservices as it is time consuming and requires significant manual effort even for conceptually easy modifications.

3.3 Need for Modifying Microservices

From the findings of §3.2, we consolidate three main use-cases that require users to modify microservices - (i) developing and modifying applications, (ii) understanding and reproducing end-to-end phenomena, and (iii) prototyping and evaluating improvements for microservice systems. We detail each use case below.

3.3.1 Use Case 1 (UC1): Design Space Exploration

Given the large design space of microservice systems, it is difficult to determine *a priori* which combination of design choices is optimal for a particular application or workload. Consequently, developers and researchers often need to explore the microservice design space by evaluating multiple concrete implementations before selecting an appropriate design point. To do so, they frequently *modify* applications—that is, change, reconfigure, or extend some aspect of the system design. A modification alters the application along one or more of the dimensions discussed in §2.1. For example, replacing the RPC framework from Thrift to gRPC changes only the instantiation layer; introducing replicated services together with a load balancer modifies both scaffolding and instantiations; and refactoring the application workflow typically changes the application logic while leaving scaffolding and instantiations untouched [106]. Ultimately, such modifications move the application from one point in the design space to another, producing a new variant of the microservice system.

Ideally, users should be able to modify a microservice application with minimal effort, regardless of the scale of the system. However, in practice, even

conceptually simple modifications often require extensive and time-consuming changes across source code, configuration files, and deployment artifacts. For instance, switching the RPC framework in an application with 30 services requires updating each individual service to use the new framework, as the change propagates across all service boundaries. Such replacements are difficult because different instantiations of the same scaffolding layer often expose incompatible interfaces, and existing systems tightly couple application workflows with both scaffolding and instantiation details.

Similarly, introducing new scaffolding components—such as enabling distributed tracing—typically requires updating all services in the system to propagate trace context and emit the required instrumentation data. Even changes to the application workflow itself can be complex, as they require carefully rebinding modified logic to the appropriate scaffolding abstractions and instantiation interfaces. As a result, modifications that are conceptually localized often become globally invasive in implementation, significantly increasing the effort required for system evolution.

3.3.2 Use Case 2 (UC2): Reproducing Emergent Phenomena

Emergent phenomena, or emergent behaviors [277], refer to aspects of a system's runtime behavior that cannot be attributed to any single service or component in isolation. Instead, they arise from the collective and often nonlinear interactions between components under a specific workload and system design. Emergent phenomena typically manifest in end-to-end properties of an application,

including performance, correctness, and reliability. Notable examples include cascading failures [310, 414, 459], tail latency effects [102], cross-system consistency violations [5], the "laser of death" phenomenon [294], and metastable failures [57, 186].

Left unchecked, emergent phenomena can lead to degraded service performance and even large-scale outages [268, 290, 294]; as a result, they have become a central focus of recent research efforts in both industry [372] and academia [57, 186, 254, 322]. In general, users require the ability to systematically elicit emergent phenomena in microservice applications in order to understand the conditions under which they arise and to characterize their impact on end-to-end system behavior.

Most microservice systems exhibit some emergent phenomena out-of-the-box, but these behaviors are often difficult to debug, understand, and reproduce on demand. Moreover, a given microservice system implementation may not exhibit the full spectrum of possible emergent behaviors, as these arise from specific combinations of workflow design, scaffolding, instantiation choices, workloads, and deployment conditions. As a result, studying emergent phenomena typically requires users to deliberately modify existing applications to construct variants that expose the desired behaviors. For example, no out-of-the-box microservice benchmark naturally exhibits cross-system inconsistency [254, 372] or metastable failures [57]; instead, researchers studying these phenomena manually modify existing systems to elicit them [125, 253].

It is not straightforward to identify a design that exhibits a desired emergent

phenomenon. Changes to scaffolding or instantiations can affect performance and error-propagation characteristics in highly nonlinear ways, making it difficult to predict the system-level impact of even seemingly minor modifications. Moreover, certain emergent phenomena only manifest under specific workload conditions, which are often not known to developers a priori and may not be reproducible without careful tuning or controlled experimentation. Overall, this makes empirical exploration of the design space essential: users may need to iteratively modify an application multiple times before identifying a variant that reliably and clearly exhibits the target phenomenon.

3.3.3 Use Case 3 (UC3): Prototyping and Evaluating Improvements

Users often need to prototype and evaluate *improvements* to microservice applications. An improvement may involve applying workflow design patterns [106], enabling new forms of scaffolding [253], or replacing existing instantiations with alternatives that provide better performance, correctness, or reliability properties [213]. Such improvements are typically motivated by system-level concerns, including scalability, end-to-end latency, and the mitigation or amplification of emergent phenomena. For example, FIRM [322] is an improvement aimed at mitigating SLO violations; it leverages distributed tracing as scaffolding and introduces additional orchestration scaffolding along with new instantiation logic to dynamically manage system behavior under load.

To develop and evaluate improvements, it necessary to modify existing appli-

cations to incorporate the improvement. This entails the same degree of manual overhead described for UC1 and UC2, but is exacerbated in two ways: first, development may require multiple iterations of design to converge on appropriate interfaces, potentially entailing repeated mutations over time integrating successive versions of the improvement; second, best practices for rigorous evaluation demand that any improvement should be evaluated across multiple, diverse applications [177], thus requiring effort to modify multiple applications, not just one.

Due to the high cost of modifying microservice applications, most research works today evaluate their techniques on only a single application (78%). Consequently, it is difficult to determine whether a proposed improvement would generalize to other applications or whether its effectiveness is specific to the chosen evaluation target. Moreover, improvements often implicitly rely on assumptions about an application's workflow, scaffolding, or instantiations, which can significantly limit their broader applicability. For example, FIRM [322] assumes a deterministic critical path for each API request, which may not hold in workflows involving concurrent execution paths or branching RPC calls, thereby restricting its applicability to such systems.

3.3.4 Need for Configure, Build, Design (CBD)

The common requirement that emerges from §3.3 is that users must iteratively Configure, Build, and Deploy (CBD) microservice systems. Across the use-cases discussed above, developers repeatedly perform cycles of configuration, imple-

mentation, and deployment, where each iteration involves selecting high-level system design choices, constructing or modifying the corresponding implementation, and then deploying the system to evaluate its behavior under realistic workloads and conditions.

3.4 Manual Effort in Modifying Microservices

Supporting microservice modifications is currently a challenging endeavour. This is because making modifications to microservice systems is a highly labor intensive task requiring significant amount of manual effort. The underlying reason for this is that microservice applications today tightly couple the various different features of the system. For example, in the ComposePost service example shown in Fig. 3.1, the microservice's workflow, scaffolding, and instantiation are mingled together to produce an implementation. Consequently, making any modification to this service requires changing the full implementation while making sure that none of the existing features are broken in the process. This lack of modularity and tight coupling in a microservice implementation makes it labour intensive for users to modify microservice applications.

Chapter 4

Blueprint Design and Implementation

Blueprint is a microservice development toolchain that supports rapid *Configure, Build, and Deploy* (CBD) workflows for microservice applications. Blueprint enables users to easily *modify* application designs and automatically generate complete, runnable, and deployable system implementations that incorporate the desired changes. Rather than manually editing generated source code, users only need to update the corresponding high-level declarations in Blueprint's abstractions, with the toolchain automatically propagating those changes throughout the implementation.

Moreover, Blueprint reduces duplicated engineering effort by encapsulating scaffolding and instantiation logic as reusable compiler extensions and plugins. These extensions can then be systematically reused across applications, allow-

ing Blueprint to automatically transform and regenerate existing systems with minimal manual effort while preserving modularity and consistency across implementations.

We designed Blueprint to address the use-cases described in §3.3. Consequently, Blueprint can serve both as a research platform for studying microservice systems and as an early-stage design-space exploration tool for developers. By enabling rapid generation and modification of system variants, Blueprint allows users to efficiently prototype, evaluate, and iterate on different architectural and implementation choices. Concretely, using Blueprint, users can:

- develop new application workflows and generate runnable systems. Instead of binding workflow code to specific scaffolding and instantiations (e.g. the choice of RPC framework), those are declared separately with 10s of LoC, and incorporated by Blueprint when it generates the variant of the system.
- modify off-the-shelf microservice applications (*e.g.* an open-source benchmark) with just a few LoC, to swap an instantiation (*e.g.* RPC framework), enable or disable scaffolding (*e.g.* replication), or change backends (*e.g.* database), or change its workflow (*e.g.* change dependencies between services).
- introduce support for new instantiations (*e.g.* an experimental RPC framework) or scaffolding concepts (*e.g.* geo-replication) and transparently apply them to existing applications; these are implemented as compiler extensions which are independent and agnostic to specific application workflows.

4.1 Approach

The key insight of Blueprint is that based on the observation that microservice applications can be decoupled into workflow, scaffolding, and instantiations, we can design abstractions that can transform the process of designing and implementing microservice systems into a HITLSDA approach. As part of this approach, Blueprint provides language abstractions for users to programmatically specify the design of the system and a novel compiler automatically converts this design into a complete, deployable system.

4.1.1 Programmatically Specifying the Creative Component

Separation of Concerns. The workflow of an application is fundamentally independent of specific implementation choices such as the RPC framework used or the presence of additional scaffolding mechanisms such as replication, tracing, or caching. However, in conventional microservice implementations, these concerns are often tightly intertwined in the application code, resulting in systems where workflow logic, scaffolding behavior, and concrete instantiations are deeply coupled. Despite this coupling, the interfaces between these components are typically narrow because they represent conceptually distinct concerns and require only limited information about one another.

Blueprint avoids this tight coupling by separating these concerns and only combining workflow, scaffolding, and instantiation logic during code generation and compilation. Rather than embedding implementation details directly into

application logic, Blueprint allows these *design aspects to be concisely declared* through its high-level abstractions: the *workflow specification* and the *wiring specification*.

4.1.2 Automatically Executing the Mechanical Component

Blueprint's compiler combines the workflow and wiring specifications to *automatically generate the system*. The compiler produces all necessary artifacts including glue code, configuration files, wrappers, scripts, deployment manifests, and other auxiliary components—to create a complete, runnable, and deployable implementation of the specified application.

Although workflow, scaffolding, and instantiations are conceptually separate concerns, in practice they manifest at multiple levels of abstraction and granularity, including application-level libraries, sidecar processes, container images, runtime services, and orchestration-framework configurations. As a result, compile-time integration is necessary for Blueprint to consistently abstract across these heterogeneous layers while preserving compatibility between generated components.

To support this, Blueprint's compiler encapsulates a broad range of system-generation responsibilities, spanning source-code generation, process orchestration, container image construction, dynamic service addressing, deployment configuration, and integration logic. This allows users to modify high-level application specifications while delegating the complex and repetitive mechanics of system construction to the compiler toolchain.

4.2 Anatomy of a Blueprint Application

A Blueprint application uses two key language abstractions. The *workflow spec* abstraction is the implementation of the application's workflow. The *wiring spec* abstraction declares the scaffolding and instantiations to apply to the workflow. We describe each in detail.

4.2.1 Workflow Spec

The basic building block of a workflow is Blueprint's *service* abstraction. The service abstraction consists of an interface with typed methods, and provide an implementation of those methods. Blueprint currently supports Go. Fig. 4.1 defines a ComposePost service from the DSB Social Media application [139] that enables callers to upload new social media posts. Method implementations can be arbitrary and import arbitrary libraries.

Blueprint's *backend* abstraction offers interfaces for different kinds of backends such as caches, databases, and queues; Fig. 4.2 depicts a simple Cache interface. Unlike for services, a backend instantiation will have method implementations provided as part of its Blueprint compiler integration, e.g. memcached will provide a memcached client.

Blueprint uses a *dependency injection* pattern on service implementations. A service can invoke the methods of another service or a backend (Line 15). These dependencies should not be directly instantiated in the workflow spec and must be received as constructor parameters (Line 11). Services do not directly incorporate any scaffolding (*e.g.* configuring an RPC server) or bind specific instan-

```go
// Service interface exposed to other services
type ComposePostService interface {
    ComposePost(ctx context, userID int64, postContent string) error
}
// Service implementation
type ComposePostImpl struct {
    postStorageService PostStorageService
    userService UserService
}
// Constructor for instantiating the service instance
func NewComposePostImpl(ps PostStorageService, us UserService) (*ComposePostService, error) {
    return &ComposePostImpl{ps, us}, nil
}
func (c *ComposePostImpl) ComposePost(ctx context, userID int64, postContent string) error {
    creator, err := c.userService.GetUser(ctx, userId)
    if err != nil{
        return err
    }
    post := Post{Creator:  creator, Text:  text}
    return c.postStorageService.StorePost(ctx, post)
}
```

Fig. 4.1: Workflow Spec for DSB Social Network ComposePost. Lines 15-18 implement the functionality shown in Fig. 3.1

```go
type Cache interface {
    Put(key []byte, value []byte) error
    Get(key []byte) ([]byte, error)
}
```

Fig. 4.2: Built-in interface for a *cache* backend component.

```go
func makeDockerSpec(spec wiring.WiringSpec) ([]string, error) {
    uc := memcached.Container(spec, "uc")
    ud := mongodb.Container(spec, "ud")
    us := workflow.Service[UserService](spec, "us", uc, ud)
    pc := memcached.Container(spec, "pc")
    pd := mongodb.Container(spec, "pd")
    ps := workflow.Service[PostStorageService](spec, "pss", pc, pd)
    cs := workflow.Service[ComposePostService](spec, "cs", ps, us)
    trace_collector := zipkin.Collector(spec, "zipkin_collector")
    applyScaffolding := func(spec wiring.WiringSpec, serviceName string) string {
        opentelemetry.Instrument(spec, serviceName, trace_collector)
        grpc.Deploy(spec, serviceName)
        goproc.Deploy(spec, serviceName)
        return linuxcontainer.Deploy(spec, serviceName)
    }
    us_cntr := applyScaffolding(spec, us)
    ps_cntr := applyScaffolding(spec, ps)
    cs_cntr := applyScaffolding(spec, cs)
    return []string{us_cntr, ps_cntr, cs_cntr}, nil
}
```

Fig. 4.3: Wiring Spec for three dependent DSB services. Zipkin tracing is enabled for all services; they are deployed in Docker containers; and communicate using gRPC. Cache and database instantiations are Memcached and MongoDB respectively. The color coding matches the color coding shown in Fig. 3.1

tiations (*e.g.* binding to a memcached client library) as these are automatically integrated later by Blueprint's compiler. Blueprint uses Go's error-handling conventions to wrap and encapsulate errors that may be introduced from scaffolding, and Go's context propagation for implementing scaffolding such as tracing.

The above abstractions are necessary for several reasons. First, the addressing scope of callee services can vary – they could be application-level instances in the same address space, or running in separate container instances in a different datacenter, requiring network calls and address translation. Second, scaffolding may interpose calls between services, *e.g.* to add functionality like tracing or to implement RPCs. Third, scaffolding may duplicate or replicate service instances in some way. In all cases, Blueprint's compiler later generates the code that instantiates, configures, and addresses services, at the application, process, and container granularities.

Blueprint includes numerous out-of-the-box open-source applications, and once a developer has implemented an application's workflow spec it only needs to be revisited if workflow logic needs to change. Changes to scaffolding and instantiations happen through Blueprint's wiring spec.

4.2.2 Wiring Spec

The *wiring specification* declares the topology of the application, applies scaffolding, and configures concrete instantiations. It is written as a Go function that is executed by Blueprint's compiler during compilation (see Fig. 4.3). During this process, the compiler invokes the specification function with an initially empty

wiring `spec` object, and the function declaratively populates this object by adding Intermediate Representation (IR) nodes corresponding to service instances, backends, scaffolding components, and deployment structure.

A typical wiring spec is concise—often only tens of lines of code—and is designed to be easily modified by developers without requiring changes to the workflow specification. This separation allows users to evolve system structure and deployment decisions independently, with Blueprint automatically recompiling the application variant and generating substantially different implementation artifacts as needed.

Within the wiring spec, developers first declare instances of services defined in the workflow specification (Lines 4,7, and 8), connect their dependencies, and instantiate required backends (Lines 2-7). The spec also defines deployment groupings such as processes and containers, thereby determining how components are partitioned across runtime boundaries.

To enable scaffolding, users apply plugin-provided functions to individual service instances (Lines 10-18). Each such call triggers the corresponding compiler plugin during compilation, which then generates or transforms code and associated artifacts. Depending on the abstraction level, scaffolding may apply to service instances, processes, or container images. For example, OpenTelemetry instrumentation (Line 11) wraps service instances using proxy classes, while Linux container scaffolding (Line 14) operates at the process level.

Overall, the wiring specification serves as a declarative bridge between high-level application structure and low-level deployment artifacts, enabling modular

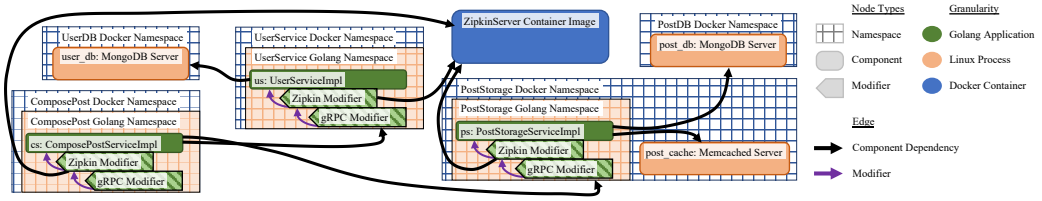


Fig. 4.4: A depiction of Blueprint's IR for the three DSB services outlined in Fig. 4.3. Node shape is based on node type, and nodes are colored according to their granularity. Edges are color-coded according to the type of relationship.

composition of instantiations and scaffolding while preserving a compact and easily modifiable representation of system design.

4.3 Intermediate Representation (IR)

The canonical representation of a Blueprint application is the compiler's Intermediate Representation (IR). Blueprint's compiler takes as input an application's workflow spec, wiring spec, and enabled compiler plug-ins. The IR of an application is a verbose and well-structured graph that represents the concrete layout and hierarchy of components along with their interactions. The IR of an application depends on both the workflow and wiring spec, and a change to just the wiring spec (e.g. to add replication) will result in a different IR. The IR is designed to support the flexibility and extensibility of the compiler. Fig. 4.4 depicts the IR graph for the wiring spec outlined in Fig. 4.3.

4.3.1 Types of IR nodes

Tab. 4.5 lists all the different types of IR nodes along with a brief description. We discuss each node type in greater detail below.

IR Node Type	Description
Application Node	Root-level IR node representing the complete application and containing all other IR nodes.
Component Nodes	Represent deployable runtime entities such as services, databases, binaries, or container images.
Namespace Nodes	Group components into coarser-grained deployment units such as processes, containers, or regions.
Modifier Nodes	Interpose functionality between components to add scaffolding such as RPC, tracing, or timeouts.
Generator Nodes	Represent components that dynamically create and manage runtime-generated instances.

Tab. 4.5: Types of IR nodes in Blueprint.

Application Node. The Application IR node is the root-level IR node that represents the full system definition and contains all the IR nodes defined in the wiring specification.

Component Nodes. Components are entities that will ultimately be deployed in the generated system; they are represented as nodes in the IR. All services defined in the workflow spec have corresponding component nodes; likewise all backends and instantiations. Component nodes can exist at different granularities, *e.g.* representing an application-level service instance, a pre-defined binary (*e.g.* a MongoDB instance), or a pre-built container image (*e.g.* a ZipkinServer). For example, Fig. 4.3 defines seven component nodes - service instantiations of UserService, PostStorageService, and ComposePostService, two mongo db container instances, two memcached container instances, and a zipkin collector container instance.

Namespace Nodes. Components of the same granularity can be grouped un-

der a namespace node to create a component of coarser granularity. For example, a Go namespace node groups together application-level instances (*e.g.* service instances) into a single Go process. Similarly, a process namespace can be grouped into a container, and a container namespace into a deployment. During compilation, namespace nodes generate runnable artifacts (*e.g.* code, container images) that instantiate their contained components. Typing on nodes ensures that namespace nodes can only contain children of a compatible granularity. Blueprint can be extended with new namespaces; *e.g.* support for georeplication would introduce a region namespace; supporting C++ workflows would introduce a Cpp namespace.

Modifier Nodes. Scaffolding can interpose edges between components, *e.g.* to modify method signatures, add proxy functionality, or enable addressing across address space boundaries. These *modifier* nodes attach to component nodes and mutate the component's edges (e.g. to introduce client side and server side code). Modifiers must be opaque to the caller component whom expects a particular method signature from the callee; thus a modifier typically comprises a client-side transformation function and a corresponding server-side function that inverts the transformation (e.g. serialization and deserialization). Each scaffolding introduces a separate modifier node. For example, in Fig. 4.4, the zipkin scaffolding introduces a zipkin modifier node and the gRPC scaffolding introduces a gRPC modifier node.

Generators. A component declared in a wiring spec might correspond to a single concrete runtime instance (*e.g.* those in Fig. 4.3), or as a result of applying

modifiers, to multiple runtime instances. For example, an autoscaling modifier might dynamically create and destroy multiple component instances at runtime. In general, *generator* nodes contain other nodes and represent instances that will be dynamically created at runtime. Generators restrict the visibility of contained nodes, since there will be multiple dynamically-generated instances of the contained nodes. Generators are typically coupled with functionality such as a load balancer to enable addressing of the dynamically created instances.

4.3.2 Encoding Dependencies

Dependencies. Services defined in the workflow specification may invoke methods on other services or components. For example, the `ComposePostService` in Fig. 4.1 invokes methods on both the `UserService` and the `PostStorageService`. In Blueprint's IR, these dependencies are represented as directed edges between component nodes. The edges capture the caller-callee relationship and encode the method signatures associated with the corresponding invocations.

Visibility and Addressing. Dependent components may execute in different processes, containers, or machines. For example, an application may be compiled as a single all-in-one process or as a deployment where each service runs in a separate container. Although an edge may exist between two components in the IR, those components may not necessarily be *visible* to each other at runtime—for instance, if a service is not wrapped with an RPC server, it cannot accept remote invocations. To capture this constraint, edges between components are annotated with their visibility level. IR nodes can also modify or extend visibil-

ity across boundaries, enabling communication when required. For example, an RPC modifier can expand the visibility of an edge to allow interaction between components running in different processes, thereby enabling cross-process or cross-machine communication without changing the underlying workflow specification.

4.3.3 Compiler Plugins

Scaffolding and instantiations in Blueprint are implemented as *compiler plugins*. Most applications use Blueprint's built-in plugins directly; however, researchers and developers prototyping new functionality, architectural mechanisms, or system improvements can extend Blueprint by implementing custom compiler plugins.

Compiler plugins integrate with Blueprint at three primary points. First, a plugin can introduce new *wiring functions* that are callable from the wiring specification to attach existing IR nodes to the applications intermediate representation (IR). For example, Blueprint's gRPC plugin provides a `Deploy` function that takes the name of a service instance to be deployed as a gRPC server and adds the corresponding IR nodes for that deployment into the application IR. Second, plugins can extend Blueprint's IR itself by introducing new node types, new modifier nodes, or augmenting existing ones with additional metadata and functionality. For example, the gRPC plugin defines dedicated client-wrapper and server-wrapper IR nodes that capture RPC-specific generation requirements. Third, plugins provide the implementation logic responsible for generating code,

configuration files, deployment artifacts, and other plugin-specific outputs. Plugins may additionally invoke external tools as part of the generation pipeline. For instance, the gRPC plugin generates service descriptions in the Protocol Buffers DSL, invokes the Protocol Buffers compiler to generate RPC and serialization code, and then produces the corresponding client-side and server-side wrapper classes that integrate the generated RPC interfaces into the application implementation.

A Blueprint compiler plugin typically implements all three integration points to cohesively provide a particular scaffolding feature or instantiation mechanism. However, not all plugins require the full set of integrations. Some plugins may only need to expose wiring functions if they neither introduce new IR node types nor generate additional artifacts. For example, Blueprint's replication plugin only provides a `Replicate` wiring function, which creates replicated copies of an already instantiated service IR node and inserts a corresponding Load Balancer IR node provided by the load-balancer plugin.

Blueprint is intentionally designed so that implementing a plugin is independent of the implementation details of other plugins. This modular plugin architecture allows plugins to be transparently composed together, enabling developers to combine multiple scaffolding features, deployment mechanisms, and implementation strategies without requiring direct coordination between plugins.

Pointers. As the workflow IR nodes correspond to the innermost node in the IR graph, the workflow node uses Blueprint's internals to specify in the wiring

spec that other plugins can transparently wrap around the workflow IR node. The workflow plugin does this through Blueprint's `pointer` library by declaring the workflow IR server node as a pointer. Subsequent plugins can then use Blueprint's pointer primitives `AddDstModifier` and `AddSrcModifier` to independently create new IR nodes that wrap around the server-side IR nodes and client-side workflow IR nodes respectively.

4.4 Compilation

Blueprint compiles an application in three steps as shown in Fig. 4.6. First, it executes the wiring spec and workflow spec to instantiate the specific IR graph representing the application and its topology (Lines 9-16). To do so, the compiler invokes the user-provided function that implements the wiring specification. For example, in Fig. 4.6, the compiler invokes the `makeDockerSpec` function (Line 12; `spec_fn = makeDockerSpec`) defined in Fig. 4.3. Next, it invokes any custom compiler passes over the generated IR to optimize, analyze, and/or modify the IR (Lines 19-21). Finally, it generates the runnable artifacts from the overall IR (Line 24). Lines 28-34 show the usage of the compilation procedure for the social network application defined in Fig. 4.1.

In this section, we explain the three compilation steps in detail along with code excerpts from Blueprint's compiler internals and the code generated for the social network application (defined in Fig. 4.1).

```go
// Convenient type definition for function that defines IR nodes in a wiring spec
type BuildFn func(wiring.WiringSpec) ([]string, error)

// Main compilation function that converts the workflow and wiring spec into an implementation
func ExecuteWiringSpec(app_name string, passes []IRAnalysisPass, spec_fn BuildFn) error {
    // Parse command line arguments
    args := parseArgs()

    // Make a new wiring spec object
    spec := wiring.NewWiringSpec(app_name)
    // Define the IR Nodes in the wiring spec
    nodesToBuild, _ := spec_fn(spec)

    // Construct the IR for the nodes that need to be initialized
    // along with all their dependent IR nodes
    IR, _ := spec.BuildIR(nodesToBuild...)

    // Run any custom IR analysis passes
    for _, pass := range passes {
        is_modified, _ := pass.Analyze(spec, IR)
    }

    // Generate system artifacts
    return IR.GenerateArtifacts()
}

// Example Usage of the compilation procedure for the social network application
func main() {
    passes := []IRAnalysisPass{PrintIRPass()}
    err := ExecuteWiringSpec("dsb_social_network", passes, makeDockerSpec)
    if err != nil {
        log.Fatal(err)
    }
}
```

Fig. 4.6: Code excerpt from the compilation procedure of Blueprint

4.4.1 Wiring & Workflow Spec to IR

Blueprint constructs the application's intermediate representation (IR) using the Builder design pattern over the wiring and workflow specifications. In this process, the compiler first executes the wiring specification (Line 12 in Fig. 4.6), including the instantiation of services defined in the workflow specification. During this execution, each compiler plugin used in the wiring specification registers an IR node type and provides an associated `Build` function, which is responsible for producing a concrete instance of that IR node at a later stage of compilation. The wiring specification ultimately returns the set of IR node identifiers that the user wishes to instantiate and generate artifacts for.

Subsequently, Blueprint constructs the root-level Application IR node, which aggregates all requested IR nodes (Line 16 in Fig. 4.6). To do so, the compiler iterates over the returned node identifiers and invokes the corresponding registered `Build` function for each node. This process incrementally constructs the full IR graph, including all services, scaffolding components, and instantiations.

A key advantage of this builder-based construction is that Blueprint does not need to explicitly resolve or infer dependencies between IR nodes during compilation. Instead, dependency management is encapsulated within plugin-defined `Build` functions and the wrapper nodes they generate, allowing the compiler to remain agnostic to inter-node relationships while still producing a fully connected and consistent application IR.

```
1  // IRAnalysisPass is a Blueprint compiler pass that operates on an ir.ApplicationNode.
2  // May optionally modify the ApplicationNode to add/remove/modify IR nodes
3  type IRAnalysisPass interface {
4      // Returns true if the ir was modified
5      Analyze(spec wiring.WiringSpec, app *ir.ApplicationNode) (bool, error)
6      Name() string
7  }
8
9  // Setters and Getters for static property values of given IR node
10 AddProperty(spec wiring.WiringSpec, node_name string, key string, value any)
11 SetProperty(spec wiring.WiringSpec, node_name string, key string, value any)
12 GetProperty(spec wiring.WiringSpec, node_name string, key string, dst any) error
```

Fig. 4.7: Blueprint support for custom passes over the Blueprint IR

4.4.2 Custom Compiler Passes

Fig. 4.7 shows the built-in support provided by Blueprint for defining custom compiler passes. Lines 3-8 specify the interface that each pass must implement in order to be considered valid over Blueprint's IR. The central entry point of this interface is the `Analyze` function, which takes as input both the wiring specification and the fully constructed Application IR node derived from it. Compiler passes can be composed into pipelines and executed in a user-specified order, as shown in Lines 19-21 of Fig. 4.6.

Supported Compiler Passes. Blueprint supports three categories of custom compiler passes: (i) analysis passes, (ii) transformation passes, and (iii) hybrid passes. An analysis pass examines the wiring specification and the Application IR to extract and annotate system-wide properties that may be useful for downstream compilation stages. To support this, Blueprint exposes APIs over the wiring specification that allow plugins and analysis passes to attach structured metadata to IR nodes. For example, an analysis pass for metastability anal-

ysis may annotate the IR with information such as retry counts, retry policies, and timeout configurations across services in the system. A transformation pass, in contrast, can modify the structure of the Application IR by adding, removing, or rewriting IR nodes. This is particularly useful for features that require synthesizing additional system components based on global analysis. For instance, a metastability analysis transformation pass may introduce additional IR nodes that generate inputs for external analysis tools such as Metafor [192], producing auxiliary artifacts alongside the compiled system implementation. Finally, a hybrid pass combines both analysis and transformation capabilities, enabling compiler extensions that first analyze the system to infer global properties and then use those insights to restructure or augment the IR. This enables more expressive compiler functionality for features that require tight coupling between system understanding and code generation.

4.4.3 IR to Implementation

Once the IR graph has been constructed, Blueprint performs a visibility validation step to ensure that all declared edges correspond to feasible interactions at runtime. In particular, it checks that any component configured to invoke another component has the necessary visibility to do so, taking into account process boundaries, container boundaries, and any scaffolding or modifier nodes that affect communication pathways.

After this validation, Blueprint proceeds to the artifact generation phase. In this stage, each node in the IR graph invokes plugin-specific logic to generate

the corresponding code, configuration, or deployment artifacts required to instantiate that component within the system. Blueprint traverses the IR graph in a structured manner, calling the appropriate plugin handlers at each node and collecting the produced artifacts. These artifacts collectively form the complete deployable system, including all service implementations, scaffolding layers, runtime configurations, and supporting infrastructure components.

4.4.3.1 Resolving Dependencies

As part of the generation process, Blueprint collects and resolves code dependencies across namespaces in the IR graph to ensure that all remote components are properly addressable. For instance, if a service invokes another service via RPC and the two components are deployed in different containers or processes, the generated implementation must include the appropriate client-side code for the target service as well as any associated modifiers introduced along the communication path.

In such cases, every IR node traversed by the communication edge is responsible for propagating and forwarding the target service's address as part of its generated artifacts. This ensures that the target service binds to a reachable address that can be dialed by the source service, and that deployment artifacts (such as container configurations) correctly expose and route the required ports. If any component along the communication path cannot establish or propagate the necessary addressability, Blueprint's compiler raises a compilation error indicating that the edge between the two services violates visibility constraints and cannot be realized in the generated deployment.

4.4.3.2 Plugin-Specific Artifact Generation

Blueprint generates code and artifacts in a hierarchical, bottom-up manner, beginning with the innermost nodes of the IR graph and progressively composing higher-level abstractions. For service nodes defined in the workflow specification, no additional code generation is required; instead, the compiler primarily resolves dependencies and gathers the necessary linkage information. For modifier nodes, the compiler invokes the corresponding plugin associated with the node type. The generated output from the preceding IR node is passed as input to the modifier plugin, enabling it to wrap, extend, or transform the existing implementation. In most cases, modifier nodes produce a struct-based implementation in the target language along with associated method definitions. When a modifier alters method signatures or introduces additional functionality, it also generates a corresponding interface that reflects the updated contract exposed by the modified component. For namespace nodes, such as Go process namespaces or Docker container namespaces, code generation involves packaging the artifacts produced by contained nodes and generating the necessary runtime scaffolding to instantiate and execute them. This may include process entry-points, container build configurations, and orchestration metadata required for deployment. This hierarchical generation process is applied consistently across all service instances defined in the wiring specification, ensuring that each layer of abstraction is systematically lowered into concrete executable artifacts while preserving the structure and dependencies specified in the IR graph.

As an example, consider the generation process for the ComposePostService

```go
type ComposePostService_OTServerWrapperInterface interface {
    ComposePost(ctx context.Context, userID int64, postContent string, traceCtx string) error
}

type ComposePostService_OTServerWrapper struct {
    Service socialnetwork.ComposePostService
    CollClient backend.Tracer
}

func New_ComposePostService_OTServerWrapper(ctx context.Context, service socialnetwork.ComposePostService
     , coll_client backend.Tracer) (*ComposePostService_OTServerWrapper, error) {
    handler := &ComposePostService_OTServerWrapper{}
    handler.Service = service
    handler.CollClient = coll_client
    return handler, nil
}

func (handler *ComposePostService_OTServerWrapper) ComposePost(ctx context.Context, userID int64,
     postContent string, traceCtx string) (err error) {
    if traceCtx != "" {
        span_ctx_config, _ := backend.GetSpanContext(traceCtx)
        span_ctx := trace.NewSpanContext(span_ctx_config)
        ctx = trace.ContextWithRemoteSpanContext(ctx, span_ctx)
    }

    tp, err := handler.CollClient.GetTracerProvider(ctx)
    if err != nil {
        span.RecordError(err)
        return
    }
    tr := tp.Tracer("ComposePostService")
    ctx, span := tr.Start(ctx, "ComposePostServiceServer_ComposePost")
    defer span.End()
    err = handler.Service.ComposePost(ctx, userID, postContent)
    if err != nil {
        span.RecordError(err)
    }
    return
}
```

Fig. 4.8: Generated code for adding opentelemetry-compatible tracing to the Compose-Post Service from the workflow

```go
type ComposePostService_GRPCServerHandler struct {
    UnimplementedComposePostService_OTServerWrapperInterfaceServer
    Service ot.ComposePostService_OTServerWrapperInterface
    Address string
}

func New_ComposePostService_GRPCServerHandler(ctx context.Context, service ot.
     ComposePostService_OTServerWrapperInterface, serverAddress string) (*
     ComposePostService_GRPCServerHandler, error) {
    handler := &ComposePostService_GRPCServerHandler{}
    handler.Service = service
    handler.Address = serverAddress
    return handler, nil
}

// Implements Runnable interface used by Namespace
func (handler *ComposePostService_GRPCServerHandler) Run(ctx context.Context) error {
    lis, err := net.Listen("tcp", handler.Address)
    if err != nil {
        return err
    }
    s := grpc.NewServer()
    RegisterComposePostService_OTServerWrapperInterfaceServer(s, handler)
    go func() {
        select {
        case <-ctx.Done():
            s.GracefulStop()
        }
    }()
    return s.Serve(lis)
}

func (handler *ComposePostService_GRPCServerHandler) ComposePost(ctx context.Context, req *
     ComposePostService_OTServerWrapperInterface_ComposePost_Request) (*
     ComposePostService_OTServerWrapperInterface_ComposePost_Response, error) {
    userID, postContent, traceCtx := req.unmarshall()
    err := handler.Service.ComposePost(ctx, userID, postContent, traceCtx)
    if err != nil {
        return nil, err
    }
    rsp := &ComposePostService_OTServerWrapperInterface_ComposePost_Response{}
    rsp.marshall()
    return rsp, nil
}
```

Fig. 4.9: Generated code for running ComposePostService as a GRPC server. (Request unmarshall and Response marshall functions omitted for brevity)

in Fig. 4.3. The generation procedure starts at the ComposePostServiceImpl node, `cs`, in Fig. 4.4. This node simply gathers the code dependencies directly from the workflow spec where it is defined. The compiler then steps outwards to the ZipkinModifier which inspects the method signature list of `cs` and generates a wrapper class that adds trace contexts to all methods. As tracing must happen before the start of the actual processing of the function call, Blueprint wraps the workflow specification of the service with a generated object that starts a new span every before calling into the function (shown in Fig. 4.8). Subsequently, as the next level in the hierarchy is the GRPC node, the gRPC plugin generates protobuf RPC message definitions for the expanded `cs` methods, invokes the gRPC compiler, then generates client and server instantiation code with GRPC-specific code including request/response marshalling and unmarshalling (shown in Fig. 4.9). To ensure that Blueprint can launch the GRPC server, the generated code implements a Run function that will be called at runtime to start the server.

The next node in the hierarchy, according to the IR, is the process node. Fig. 4.10 shows the generated artifact for this node, which leverages Blueprint's generated runtime namespace. The resulting process-level artifact follows a builder-based structure for constructing the runtime process namespace. As part of this pattern, the generated code first initializes required environment configuration such as bind and dial addresses (Lines 9-14), then declares all objects that must be instantiated within the process namespace (Lines 16-27), and finally defines and registers builder functions for each generated object corresponding to IR nodes contained in that namespace (Lines 29-47). The `main` function subsequently con-

```go
func New_compose_proc(name string) *NamespaceBuilder {
    b := golang.NewNamespaceBuilder(name)
    set_compose_proc_Args(b)
    set_compose_proc_Instances(b)
    set_compose_proc_Definitions(b)
    return b
}

func set_compose_proc_Args(b *NamespaceBuilder) {
    b.Required("composepost_service.grpc.bind_addr")
    b.Required("post_storage_service.grpc.dial_addr")
    b.Required("user_service.grpc.dial_addr")
    b.Required("zipkin.dial_addr")
}

func set_compose_proc_Instances(b *NamespaceBuilder) {
    b.Instantiate("post_storage_service.grpc_client")
    b.Instantiate("zipkin.client")
    b.Instantiate("post_storage_service.client.ot")
    b.Instantiate("post_storage_service.client")
    b.Instantiate("user_service.grpc_client")
    b.Instantiate("user_service.client.ot")
    b.Instantiate("user_service.client")
    b.Instantiate("composepost_service")
    b.Instantiate("composepost_service.server.ot")
    b.Instantiate("composepost_service.grpc_server")
}

func set_compose_proc_Definitions(b *NamespaceBuilder) {
    b.Define("composepost_service", func(n *Namespace) (any, error) {
        var postStorageService socialnetwork.PostStorageService
        if err := n.Get("post_storage_service.client", &postStorageService); err != nil {
            return nil, err
        }

        var userService socialnetwork.UserService
        if err := n.Get("user_service.client", &userService); err != nil {
            return nil, err
        }

        return socialnetwork.NewComposePostServiceImpl(n.Context(), postStorageService, userService,
      uniqueIDService, mediaService, textService, homeTimelineService)
    })

    // ......
    // Omitted (for brevity) build definitions for nodes instantiated in set_compose_proc_Instances
    // ......
}
```

Fig. 4.10: Generated Code for Compose Post Process Node that uses Blueprint's runtime namespace for building instances of generated object definitions

structs the runtime namespace by invoking these builder functions to instantiate all objects and then executes the system by calling `Run` on those objects that implement the `Runnable` interface.

At the highest level of the hierarchy, the final node is the linuxcontainer node. Unlike lower-level nodes, this node does not generate Go code artifacts; instead, it produces a Dockerfile that encapsulates and executes the process-level artifact. By default, Blueprint aggregates all linuxcontainer-generated outputs into a single Docker Compose specification, enabling streamlined building, deployment, and orchestration of the resulting containerized system.

4.4.3.3 Runtime Namespaces

For instantiating and running the generated objects, Blueprint generates *runtime namespaces*. Fig. 4.11 shows an excerpt of the Blueprint-generated code for runtime namespaces. Line 1 describes the type of the constructor function, and Lines 2-4 describe the `Runnable` interface that generated objects should implement if they want to launch run functions. The runtime namespace provides a `Get` function (Lines 21-53) that builds the object for the specified node by calling the registered constructor function for that object (Line 23-27) and launches the Run function if the constructed object implements the `Runnable` interface in a separate goroutine[1] (Lines 29-47). To ensure safe exit and cleanup of resources, the runtime namespace combines all launched goroutines in a wait group which can then be awaited upon (Line 19) and properly shutdown (Lines 14-17) by the main thread of the execution.

[1]lightweight thread managed by Go runtime

```go
type BuildFunc func(n *Namespace) (node any, err error)
type Runnable interface {
    Run(ctx context.Context) error
}

type Namespace struct {
    name       string
    buildFuncs map[string]BuildFunc
    // ... omitted fields
    wg         *sync.WaitGroup
    parent     *Namespace
}

func (n *Namespace) Shutdown(awaitCompletion bool) {
    n.cancel()
    if awaitCompletion { n.Await() }
}

func (n *Namespace) Await() { n.wg.Wait() }

func (n *Namespace) Get(name string, receiver any) error {
    if build, exists := n.buildFuncs[name]; exists {
        built, err := build(n)
        if err != nil {
            slog.Error(fmt.Sprintf("%v error building %v", n.name, name))
            return err
        }
        // Execute the Run function if the built object implements the Runnable interface
        if runnable, isRunnable := built.(Runnable); isRunnable {
            n.wg.Add(1)
            if n.parent != nil {
                n.parent.wg.Add(1)
            }
            go func() {
                err := runnable.Run(n.ctx)
                if err != nil {
                    slog.Error(fmt.Sprintf("%v error running node %v: %v", n.name, name, err.Error()))
                    n.cancel()
                }
                n.wg.Done()
                if n.parent != nil {
                    n.parent.wg.Done()
                }
            }()
        }
        return nil
    }
    // Support hierarchical namespaces
    if n.parent != nil {
        return n.parent.Get(name, receiver)
    }
    return fmt.Errorf("%v unknown %v", n.name, name)
}
```

Fig. 4.11: Generated Code excerpt for a runtime namespace

4.5 Executing Blueprint Applications

Blueprint's abstractions reduce the manual effort required to generate concrete implementations of microservice systems. Generating runnable systems is only the first step in the broader design-space exploration process: developers must subsequently validate functionality and execute representative workloads to measure system behavior and performance under realistic operating conditions. However, as developers iterate across different designs and architectural choices, they would traditionally need to repeatedly re-implement workloads, tests, and client-side logic to remain compatible with the changing system interfaces and communication libraries.

To automate this process, Blueprint provides plugins for generating end-to-end tests and workload generators compatible with the generated implementation without requiring manual reimplementation effort. Using these plugins, users specify black-box tests and workload-generation processes directly as part of the workflow specification, which are then instantiated through the wiring specification to automatically generate the artifacts necessary for testing and executing Blueprint applications. This enables developers to rapidly evaluate different system variants while avoiding duplicated engineering effort across iterations of the design exploration process. Further details about these plugins are provided in §A.

4.6 Prototype

Blueprint's prototype is implemented in 30K lines of code. Blueprint provides first-class support for Go workflow specs. We selected Go because it is designed specifically for high-performance RPC services, and has convenient mechanisms for handling concurrency, errors, and context propagation. Blueprint is not constrained to Go; the abstractions of Blueprint support extension to multiple languages. Blueprint is open-source and available for public use at `https://github.com/Blueprint-uServices/blueprint`.

Chapter 5

Evaluation

We evaluate the efficacy of Blueprint by answering the following questions:

- Does Blueprint reduce effort for design space exploration (UC1)? (§5.2)
- Can Blueprint help reproduce emergent misbehavior in microservice applications (UC2)? (§5.3)
- Does Blueprint reduce the effort for prototyping improvements (UC3)? (§5.4)
- Are Blueprint-generated systems realistic? (§5.5)
- Is Blueprint easy to extend with new scaffolding and instantiations? (§5.6)
- What is the cost of Blueprint's abstractions? (§5.7)

5.1 Experimental Setup

All experiments use a cluster comprising eight machines, each with four Intel Xeon E7-8857V2 processors, 48 cores and 1.5 TB RAM. We deploy each service

in a separate container. We use a simple open-loop workload generator that can be configured to exercise APIs of the generated system with a specified request rate and API distribution; this runs on a separate machine.

For our experiments, we modify the Blueprint implementations of DSB Hotel-Reservation and DSB SocialNetwork; for readability we refer to these simply as BP-DSB-HR and BP-DSB-SN.

5.2 UC1: Design Space Exploration

Reducing Implementation Effort. To demonstrate that Blueprint makes it easy to implement realistic microservice systems not specifically designed for our evaluation, we have re-implemented five existing microservice applications in Blueprint. Tab. 5.1 shows the LoC needed to implement the workflow spec and wiring file for each application in Blueprint. We compare the LoC needed to those in the original implementations. Blueprint reduces the code footprint by 5–7× for each application by eliminating the need to implement scaffolding and instantiations alongside workflow code. In the original implementations, scaffolding was tightly coupled with workflow code, thus inflating the amount of code that a user needed to write. By decoupling the workflow specification from the scaffolding code and moving scaffolding code generation to the compiler, Blueprint reduces the volume of code required to implement a workflow. One beta user of Blueprint noted that this decoupling also made it easier for them to understand and implement the workflow specification.

Changing workflow specs. We compare the LoC required to make a change

to the design of the application in the original system and compare that to the effort to implement the same change in the Blueprint implementation of the application. In pull request #101 of DSB SocialNetwork [106], the authors inverted caller-callee relationships between `ComposePostService` and `TextService`, `UniqueIDService`, `UserService`, and `MediaService` to improve application performance. They modified 5,140 LoC. We implemented the same change in the Blueprint version of the system by modifying 288 LoC of workflow spec, and 7 LoC of wiring spec — a 17× reduction. The substantial difference in code changes arises due to Blueprint's separation of concerns: in the original implementation, changing interfaces between services required changing scaffolding and instantiation code, which was tightly coupled with workflow code. However, with Blueprint, scaffolding and instantiation code changes are handled by the compiler, and only the workflow specification required manual changes.

Changing scaffolding and instantiations. Blueprint makes it easier to enable or disable new scaffolding in an application. Based on our survey of modifications of microservice systems, we found that a popular change in existing microservice systems is to enable or disable tracing [19, 44, 178, 196]. For example, disabling tracing in a fork of DSB SocialNetwork required 418 LoC [196]. In contrast, the same change required 5 LoC wiring spec change for the Blueprint implementation of DSB Social Network. This small wiring spec change automatically removes ∼2 KLoC from the generated system, including all tracing source code modifications and configuration files needed to enable tracing in the runtime.

System	Orig. (LoC)	Blueprint (LoC)		Reduction
		Spec	Wiring	
DSB SocialNetwork	8 209	1 478	57	5.4×
DSB Media	7 794	1 401	42	5.4×
DSB HotelReservation	5 160	679	63	7.0×
TrainTicket	54 466	9 639	166	5.6×
SockShop	13 987	2 261	40	6.1×

Tab. 5.1: Lines of code of original and Blueprint implementations of popular open-source microservice systems

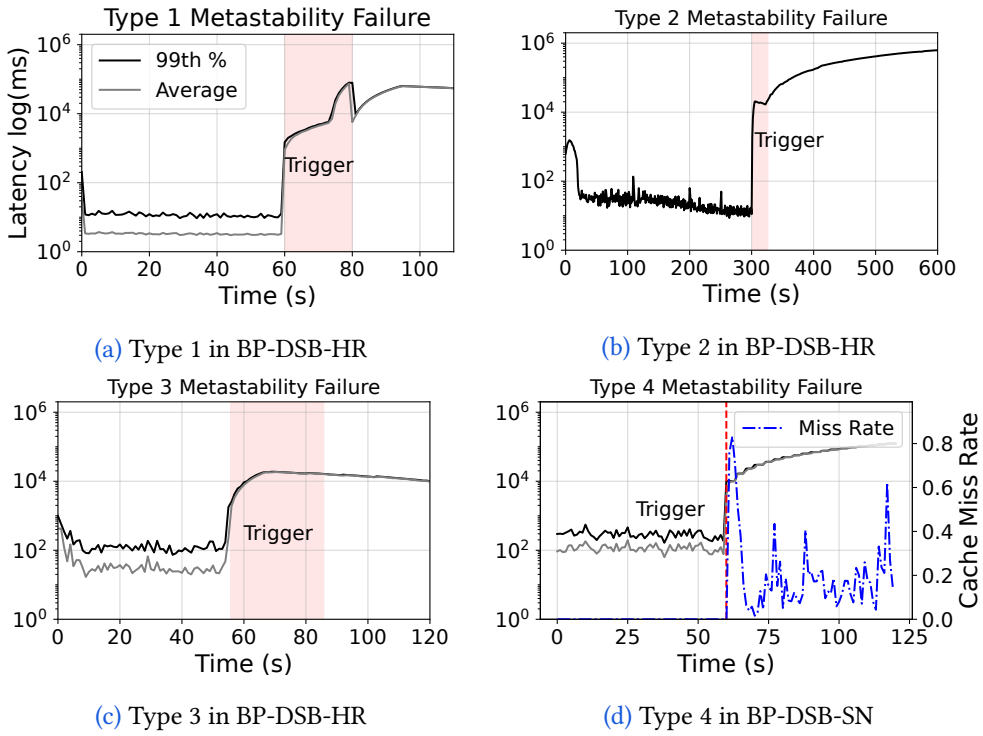
5.3 UC2: Eliciting Emergent Phenomena

5.3.1 Case Study I: Metastability Failures

We adapt BP-DSB-HR and BP-DSB-SN to showcase the four different kinds of metastability failures [186]. **The required changes described below are to the wiring spec and amount to at most 3 LoC per failure type.**

Load spike trigger workload amplification (Type 1). We modify BP-DSB-HR to add a 500 ms timeout to every inter-service RPC. We also modify the services to re-try up to 10 times on error. We start with a 10 K requests/s (Rps) workload for 60 s, then increase to 30 KRps for 30 s, and then revert to 10 KRps. Fig. 5.2a shows the mean and 99^{th} percentile service latencies over time. At the 1-minute mark, the sudden workload increase triggers the majority of requests to time out, in turn causing more requests to be generated due to retries. This trigger keeps the system in a metastable state and even after decreasing the load, the system does not recover to a stable state.

Load spike trigger capacity degradation amplification (Type 2). To induce



(a) Type 1 in BP-DSB-HR

(b) Type 2 in BP-DSB-HR

(c) Type 3 in BP-DSB-HR

(d) Type 4 in BP-DSB-SN

Fig. 5.2: Four metastability failure types [186] demonstrated in the BP-DSB-HR and BP-DSB-SN.

this type of metastability failure, the authors [186] limited the maximum service heap size. We experiment with BP-DSB-HR's `ReservationService`. As Go offers no direct control over heap size, we instead increase the garbage collection (GC) frequency by causing the GC to trigger whenever the heap is 75% full instead of the default 100% (for this, we set the environment variable `GOGC` to 75). We run a 20 KRps workload for 10 mins; at the 5 min mark we introduce low-level CPU contention for 30 s using FIRM's anomaly injector [322]. Fig. 5.2b shows mean service latency over time. Here, the CPU contention acts as a trigger by increasing the GC duration, which results in frequent stop-the-world GC phases, caus-

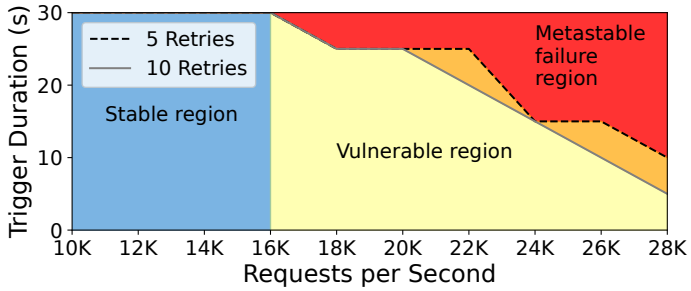


Fig. 5.3: Metastability Vulnerability Analysis for BP-DSB-HR.

ing other requests to start timing out and generate more retries. This metastable state also persists after the CPU contention disappears.

Capacity decreasing trigger workload amplification (Type 3). To induce this metastability failure, we first modify BP-DSB-HR to have 1 s timeouts and up to 10 retries on every RPC. We run a 24 KRps workload for 2 mins. After 1 min, we inject low-level resource contention with FIRM's [322] anomaly injector for 30 s to decrease available CPU capacity. Fig. 5.2c shows the mean and 99^{th} percentile service latencies over time. CPU contention starting at 1 min causes timeouts leading to retries that overload the system and keep it in a metastable state after CPU contention disappears.

Fig. 5.3 illustrates vulnerability for different request rates, trigger durations, and maximum retries. At higher request rates, even a short trigger can cause the system to move into a metastable failure state. In contrast, at lower request rates, short triggers only cause transient issues. Fewer retries only minimally increase the tolerable trigger duration.

Capacity decreasing trigger capacity degradation amplification (Type 4).

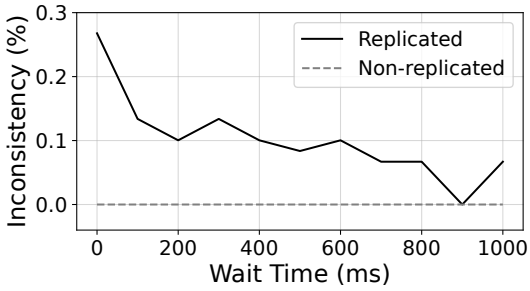


Fig. 5.4: Cross-system inconsistencies arise in BP-DSB-SN when enabling replication.

We modify BP-DSB-SN with an internal 1 s timeout and up to 10 retries. We run this experiment in two phases. First, we fill the `UserTimelineCache` with all content from the `userTimelineDatabase`. In the second phase, we run a 3 KRps workload for 2 mins. At 1 min, we flush the `UserTimelineCache` which then causes future requests to query the database. Fig. 5.2d again shows the mean and 99^{th} percentile service latencies along with the observed cache miss rate. The cache flush at 1 min overloads the database and causes cascading timeouts and retries. This overload prevents the cache from repopulating and keeps the system in a metastable failure state.

In all cases, Blueprint enables rapid exploration of different designs, such as adding timeouts and retries, which, in turn, enables the reproduction and analysis of metastable failures.

5.3.2 Case Study II: Cross-System Inconsistency

We add replication scaffolding to BP-DSB-SN to elicit cross-system inconsistencies. Cross-system inconsistencies occur when delays in synchronization of replicated data stores such as databases and caches cause read requests for an ob-

ject to return incorrect results. By default BP-DSB-SN has no replication, so we add replication to `userTimelineDatabase` and `UserTimelineService`, and modify the `GatewayService` to use the replicated version of the service. These changes only touch 4 LoC in the wiring spec.

We compare the replication-enabled BP-DSB-SN to the initial Blueprint BP-DSB-SN. We use a 100 Rps workload consisting of 100 % `ComposePost` requests. For each successful request, we read the user timeline of the post creator after a wait time ranging between 0 s and 1 s at 100 ms intervals. A response without the new post is a cross-system inconsistency. Fig. 5.4 shows the measured fraction of inconsistencies with increasing wait times for the original BP-DSB-SN and the replicated version. As expected, the non-replicated version always reads consistent results, whereas the replicated version incurs a small fraction of inconsistencies [38].

Overall, this case study demonstrates how Blueprint applications are amenable to change and enable users to modify existing applications to reproduce emergent phenomena with minimal effort.

5.4 UC3: Prototyping Improvements

In this section we evaluate how amenable Blueprint is for supporting prototyping and evaluation of improvements in two ways: (i) reproducing the prototyping required for a solution performed in existing research; and (ii) prototyping a new solution for an emergent phenomenon.

Reproducible Research. To understand how Blueprint can aid researchers in

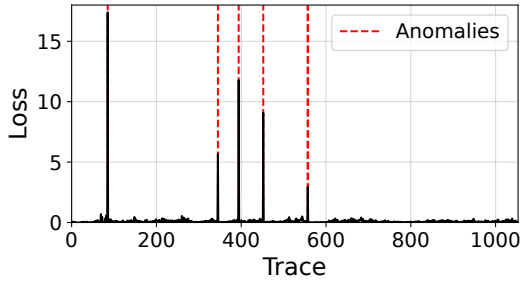


Fig. 5.5: Blueprint's reproduction of Fig. 6 from the Sifter paper [234] in SocialNetwork.

making changes, we select a mutation that was manually added by researchers to an existing microservice application, and reproduce that mutation in the Blueprint implementation of the application. Concretely, Sifter [234] manually mutates the DSB Social Network application to add X-Trace [131]. X-Trace is a distributed tracing framework, but it does not conform to OpenTelemetry APIs and cannot reuse the existing Jaeger instrumentation of DSB SocialNetwork. Consequently, the authors of Sifter manually extended DSB SocialNetwork to add X-Trace support. This comprised 1,289 LoC changed over a 2 week period, based on commit timestamps.

We implemented the same change in Blueprint, which required (1) extending Blueprint's compiler to support X-Trace (a 1-time task); and (2) enabling X-Trace for the Blueprint SocialNetwork application. The latter required 3 LoC change to the wiring spec of the SocialNetwork application. This reduction occurred because the vast majority of code to support X-Trace is templatable scaffolding code which can be easily incorporated in the compiler. Implementing X-Trace within Blueprint's compiler required 433 LoC.

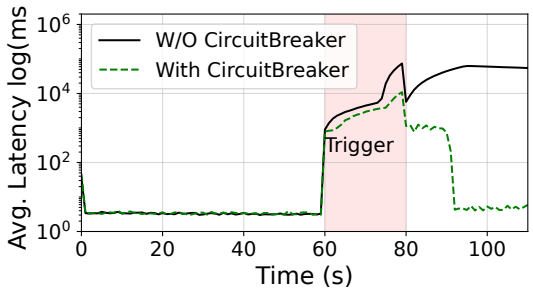


Fig. 5.6: Prototype Solution for Type 1 Metastability failure in HotelReservation.

To evaluate if Blueprint's modifications to systems yield the same results as the modifications to original systems, we reproduced Figure 6 from the Sifter paper [234] using the Blueprint-generated SocialNetwork application. In the original experiment, the authors recorded the loss and sampling probability for a sequence of 1000 ComposePost API requests, and at five separate instances they had inserted anomalous requests. Similarly, in our experiment we generated anomalous requests with the Blueprint generated DSB-SN and repeated the above experiment. In Fig. 5.5, the spikes of high probability of selection correspond to the anomalous requests. This shows that Blueprint generated systems can reproduce the same results as the original systems.

Prototyping New Solutions. In this experiment we demonstrate how Blueprint can help in prototyping and integrating novel solutions in applications. We particularly focus on the Type 1 Metastable Failure first introduced in §5.3.

To address the Type 1 Metastable Failure, we prototype a solution based on *circuit-breakers*. A circuit-breaker prevents new requests from being sent if the moving-average failure rate of requests is higher than a specified thresh-

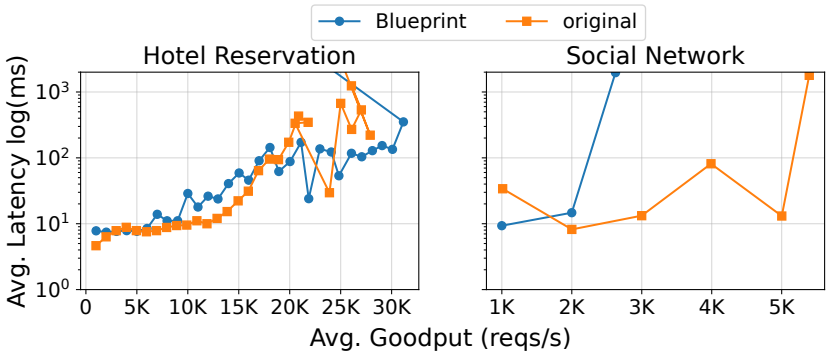


Fig. 5.7: Performance comparison of Blueprint systems with their original counterparts

old. We implement a circuit-breaker feature as a Blueprint plugin. This required 126 LoC. Enabling circuit-breakers for the HotelReservation application required only 2 LoC change to its wiring spec.

Fig. 5.6 shows how adding circuit-breakers can potentially prevent the system from going into a metastable failure state. The circuit-breaker-enabled version of the application experiences the same latency increase at $t = 60$, but due to the circuit-breaker triggering, the application is able to avoid entering a metastable failure state and returns to normal at $t \approx 90$. Overall, this example demonstrates how Blueprint can be useful in prototyping novel solutions for phenomena.

5.5 Generating Realistic Systems

We now evaluate Blueprint's ability to generate real systems useful for meaningful evaluation. For this we measure and compare the performance of two of the Blueprint-generated systems applications above to that of the original systems.

HotelReservation. We compare the performance of the Blueprint implementa-

tion of HotelReservation to the original DSB implementation. The original DSB HotelReservation application is implemented in Go, enabling a direct comparison between it and the Blueprint-generated application. To exercise the systems, we run a mixed workload of 60% `hotels`, 38% `recommendations`, 1% `user`, and 1% `recommend` requests. We run the workload for different request rates ranging from 1 Krps to 30 KRps for a duration of 1 min each. Fig. 5.7 shows the latency-throughput profile of both systems. The Blueprint generated system has a similar performance to the original manually implemented system.

SocialNetwork. We also compare the performance of the Blueprint implementation of SocialNetwork to the original DSB implementation. The original DSB implementation uses a mix of C++ and Lua with an nginx gateway web server whereas the Blueprint implementation uses Go's default HTTP web server. To exercise both systems, we run a mixed workload of 60% `ReadHomeTimeline`, 30% `ReadUserTimeline`, and 10% `ComposePost` requests. We run the workload for different request rates ranging from 1 KRps to 6 KRps for a duration of 1 min each. Fig. 5.7 shows the latency-throughput profile for both systems under the aforementioned workload. In this scenario, the original system outperforms the system generated by Blueprint. We attribute this to two compounding factors. First, the original system is implemented in C++ whereas the Blueprint version is in Go. Go incurs garbage collection overhead and relies on different libraries for many core microservices building blocks. Second, the Blueprint implementation does not use any Redis-specific specialized array operations. This illustrates a drawback of Blueprint– it requires interacting with backends through common

Backend	Interface
Cache	37
NoSQLDB	119
RelDB	42
Queue	29
Tracer	58
Log	90
Metric	57

Tab. 5.8: Lines of Code required for adding a backend interface.

Type	Instantiation	Impl	Compiler
Cache	In-Memory Cache	83	106
Cache	Redis	86	203
Cache	Memcached	96	208
NoSQLDB	MongoDB	288	140
NoSQLDB	In-Memory NoSQLDB	3658	106
RelDB	MySQL	91	140
RelDB	Sqlite	55	106
Queue	RabbitMQ	50	111
Queue	In-Memory	65	106
Tracer	Jaeger	35	232
Tracer	Zipkin	35	216

Tab. 5.9: Lines of Code required for adding a new instantiation.

interfaces.

Overall, these results illustrate that Blueprint can generate microservice systems whose performance compares closely to that of handwritten systems.

5.6 Extensibility of Blueprint

Adding backends and instantiations. Tab. 5.8 shows the LoC in the interface of various Blueprint backends and Tab. 5.9 shows the LoC for the instantiations currently available for each backend. Adding a new backend generally requires $\approx$100 LoC for implementing the lightweight interface. Adding an instantiation also requires a small amount of code ($<$200 LoC) in the compiler. Each of these is a one-time effort, that can be leveraged by subsequent Blueprint applications.

Instantiations of certain backends require more code than others. For example, instantiations of the NoSQLDB backend can require a lot more code to support the full query semantics of MongoDB, as is the case for the In-Memory NoSQLDB which required $3.7K$ LoC.

Adding new plugins. Blueprint's modifier abstraction allows users to add new scaffolding. Tab. 5.10 shows the plugins currently available in Blueprint and the LoC in their implementations. Some plugins require changes only in the Blueprint toolchain whereas others also require an additional runtime library component. The LoC vary across features from 50 to around 1.1K.

Adding new plugins is harder than directly implementing the scaffolding hard-coded within an application. Nonetheless, this addition is a one-time cost that amortizes the effort of reimplementing scaffolding in all later systems.

Plugin	**Compiler** (LoC)	**Stdlib** (LoC)
Retry (4 algorithms)	892	0
Timeouts	242	0
OpenTelemetry	732	45
HealthChecker	288	0
Replication	50	0
ClientPools	317	220
X-Trace	868	181
CircuitBreaker	233	0
LoadBalancer	151	19
GoVector	605	125
Workload	238	232
Runtime Namespace	0	439
GRPC	1238	0
Thrift	1190	0
HTTP	544	0
E2E Tests	646	70
Docker	1209	0
Docker-Compose	719	0
Kubernetes	728	0

Tab. 5.10: Lines of code required for adding a new compiler plugin.

System Name	Gen Time(s)	Num Services
DSB-SN	1.172	28
DSB-MM	1.698	33
DSB-HR	1.281	18
TrainTicket	3.723	67
SockShop	0.925	14
Alibaba-TraceSet	707	2882

Tab. 5.11: Time taken by Blueprint for generating the system including services, caches, databases, queues, tracers

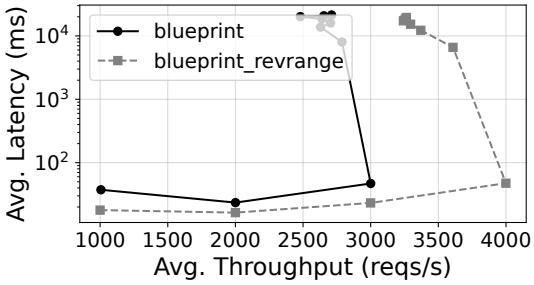


Fig. 5.12: DSB-SN cache choice exploration

5.7 Cost of Blueprint

Compilation time. Tab. 5.11 shows the time taken by Blueprint to generate systems. Blueprint can generate small to medium sized system within seconds. As there are no existing large open-source microservice systems, we generated a large-scale microservice application using the Alibaba service topology in the Alibaba trace dataset [257]. For this, we omitted the caches and databases and only focused on stateless services. In total, the resulting workflow and wiring spec had 2.8K service instances. To generate a variant implementation, Blueprint took around 12 minutes. Overall, the compilation time is proportional to the number of service instances in the wiring spec and the density of the service topology. These results demonstrate that Blueprint enables researchers and practitioners to quickly (re-)compile applications, thus supporting rapid Configure, Build, and Deploy cycles. These results may be further improved through compiler optimizations and incremental compilation.

Cost of abstractions. For each type of backend supported by Blueprint, services interact with the backend through a generalized interface. The interface

is selected such that backend instances can be opaquely reconfigured. Yet many backends do provide broader interfaces with specialized operations that are more efficient for certain workloads. Blueprint's approach discourages services from using such operations, in the interest of reconfiguration.

To demonstrate the impact of using more general but less efficient APIs, we implement an extended Cache interface that provides access to Redis' specialized cache operations. We use this extended interface in the SocialNetwork application, we execute a `100% ReadHomeTimeline` workload for 1 minute for request rates ranging from 1 KRps to 6 KRps. With the extended interface, the application observes a 33% increase in throughput as shown in Fig. 5.12.

Chapter 6

Related Work

6.1 Tools for Microservice Generation

Several existing tools support generation of microservice systems by providing a DSL or some other programming language [6, 114, 148, 204, 280, 313, 325, 327–329, 386–388, 396, 404, 429]. However, the tools are designed for one-shot generation of microservices and select a single dimension to provide flexibility. The most prominent example is Google's ServiceWeaver [148], which provides flexibility along the deployment dimension allowing users of the tool to deploy the same application as a suite of microservices or as a monolithic application. However, like the other tools, ServiceWeaver is a poor fit CBD use cases that require modifying the application along dimensions other than the deployment dimension.

A number of existing tools provide complete DSLs for specifying microser-

vice architectures, but they often lack extensibility mechanisms for integrating new infrastructure capabilities or supporting additional architectural features. For example, Jolie [280] is a service-oriented programming language that separates the program behaviour and deployment of services in a manner similar to Blueprint. However, Jolie provides limited support for reusable infrastructure abstractions and requires developers to adopt a dedicated programming language for implementing service logic. Similarly, JHipster [204] offers the JHipster Domain Language (JDL) along with extensive tooling for generating Java-based microservice applications. Despite its strong infrastructure support, JHipster is tightly-coupled to the REST-based communication paradigm and does not readily support RPC alternatives for communication. In addition, extending the framework with new infrastructure components or platform features is difficult as JHipster is tightly integrated with Silvera [396] provides a DSL for generating microservice systems. Silvera provides predefined generators and infrastructure assumptions, but it does not offer a systematic mechanism for integrating new infrastructure services, communication paradigms, or architectural transformations. In contrast, Blueprint is a broader architectural framework aimed at supporting extensible, configurable, and infrastructure-aware microservice system generation.

Other frameworks focus less on architectural specification and focus more on simplifying the integration of reusable infrastructure components into microservice applications. For instance, SpringBoot [390] and Dapr [1] provide abstractions for common infrastructure concerns such as communication, state

management, and deployment support. These abstractions separate out reusable infrastructure components from the core implementation of the application, allowing users to select infrastructure building blocks that can be applied to an application. However, the SpringBoot framework makes binding choices along the deployment and communication dimensions of the microservice design space making SpringBoot a poor fit for CBD use cases. Dapr offers greater flexibility by allowing developers to switch between communication and deployment components, but such transitions must still be performed manually, increasing developer effort. In contrast, Blueprint allows users to easily specify the various architectural design components and automatically generates a complete, runnable system while requiring minimal effort.

6.2 Open-Source Microservice Benchmarks

Microservice benchmark suites have gained considerable popularity in recent years [139, 393, 418, 468]. All of the systems provided by these benchmarks serve as representative examples of the class of microservice systems currently deployed at large cloud providers. However, each system corresponds to only one concrete point in the design space and as they were not designed to be configurable, they are inadequate for tasks that would require exploring the design space of microservices.

Instead of making rigid choices with regards to the design space of microservices, Blueprint serves the general-purpose by allowing the users to decide the concrete point in the design space they want their generated system to repre-

sent. This allows the users to more efficiently explore the various design choices in the design space of microservice systems.

6.3 Configurability & Reusability

Two of the key features that enables Blueprint to quickly reconfigure applications are the notions of reusability and dependencies. First, the idea of reusability has been inspired from the The Flux OSKit [132]. Similar to Flux OSKit's suite of reusable OS components, Blueprint also uses commonly used microservice libraries and components as reusable building blocks for microservice applications. In contrast, Blueprint does not require glue code to be handwritten for each component for each new application, instead relying on its compiler abstractions and code generation to correctly handle composition of components. Second, Blueprint uses Dependency Injection [320] to correctly generate applications by not allowing instantiation of dependent components in the workflow specification of applications. The dependencies of any given service in a Blueprint application are generated at compilation time.

Part III

Online Specialization of Systems with Iridescent

Chapter 7

Specializing Cloud Systems

In the previous part of the thesis, we showed how a HITLSDA technique can reduce the manual effort required to build and explore the architectural design space of modular cloud systems. In this part of the thesis, we show how another HITLSDA technique can take an implementation of a system and dynamically specialize, optimize, and adapt it to different deployment settings to achieve improvements in end-to-end system performance.

7.1 Specialization is Effective but Complex

Specializing system implementations to workload characteristics and hardware can significantly improve performance and efficiency [9, 50, 107, 127, 146, 147, 211, 213, 219, 272, 279, 305, 347, 368, 392, 394, 415, 435]. To achieve these benefits for particular hardware and workload combination, system developers manually modify the *system code* and recompile to benefit from the compile-time optimiza-

tions enabled by specialization [308, 428].

There is a long line of prior work establishing the performance and efficiency benefits of code specialization, particularly in systems code. For example, interpreted languages may generate specialized instances of functions with constant parameters and optimize them accordingly [99, 251]. This is a general optimization that is typically performed automatically and transparently by the runtime.

Other optimizations are more system-specific and target concrete concerns, such as inlining small table lookups in software network functions as conditional statements [272]. Like most compiler optimizations, these approaches are typically guided by simple heuristics over local metrics (e.g., frequency of parameter values or table size), often based on developer intuition.

However, specializing system code comes at the cost of generality. A system that is heavily specialized to workload and hardware either performs poorly outside of this regime, or completely fails. As a result, developers today must carefully navigate this specialization-generalization tradeoff and optimize for the most common hardware and workload settings.

However, even selecting an implementation that provides a good balance in the specialization-generalization tradeoff *a priori* is difficult for multiple reasons. First, the workload and platform are often unknown at development time, as systems are often deployed to different hardware than on which they were developed. Second, workload and platform often change at runtime. A system might migrate to newer deployments over time. Application workloads rarely remain fixed throughout the application lifetime. Third, even for a known workload and

platform, predicting the performance of a modified system remains challenging. Modern system performance is dictated by emergent behaviors arising from the complex intra-system interactions and interactions between the system and the workload. To make matters worse, he same system and workload configuration can exhibit different performance characteristics across different hardware platforms.

As a result, making optimal system code specialization decisions is fundamentally an iterative process: run the system, profile it, adjust the code, and repeat. Unfortunately, this process is also extremely laborious for developers and is therefore rarely worth the cost and complexity in practice.

7.1.1 A Case for Code Specialization

We define code specialization as the process of generating optimized versions of a program by exploiting information that becomes known only at runtime. Rather than compiling a single fully general implementation that must handle all possible inputs, configurations, and execution environments, a specializing system treats certain runtime values or program properties as constants once they are observed. The compiler can then partially evaluate the program with respect to these known values, producing a specialized version of the code that removes unnecessary generality and exposes optimization opportunities that would otherwise remain hidden.

These specialized implementations can achieve substantial performance gains because many compiler optimizations become significantly more effective when

dynamic uncertainty is eliminated. Constant propagation, dead-code elimination, loop unrolling, vectorization, instruction selection, memory-layout optimization, and kernel fusion all benefit from having precise runtime knowledge. For example, if a loop bound becomes fixed after specialization, the compiler may fully unroll the loop and eliminate bounds checks; if a tensor shape is known, memory accesses can be reorganized to improve cache locality and utilization of vector instructions. Specialization also reduces branching overhead and enables more predictable execution behavior, which improves both CPU pipeline efficiency and accelerator utilization.

7.1.2 Illustrative Example

We illustrate the benefits and challenges of specialization with an example system: a server executing square matrix multiplications for clients. To optimize cache locality we use a blocked matrix multiplication algorithm [70, 430, 431]. We identify one key workload parameter, N, the matrix size, and one key configuration parameter, B, the block size.

Code specialization improves performance. Both parameters offer a substantial opportunity for optimization through specialization. Unsurprisingly, we find fixing B as a compile-time constant instead of leaving it a variable improves throughput by up to $6.5\times$, by enabling the compiler to unroll and vectorize the inner loops. Note that B is an internal parameter that may affect performance, but any valid choice results in correct behavior with every workload. Similarly, assumptions about the workload, N, can also simplify the algorithm, e.g. by as-

Machine / Workload	N=1024	N=256	N=64
IceLake	32	32	32
IvyBridge	16	16	4
CoffeeLake	32	4	4
AlderLake-p	32	4	2
AlderLake-e	64	4	4

Tab. 7.1: Optimal value of block size, B, for our block matrix multiply, across 5 hardware platforms and 3 workloads.

suming that N is a multiple of B we avoid the need for copying and zero-padding partial blocks.

Optimal configuration depends on workload and HW. We now compare different block sizes for different workloads (matrix sizes) on 5 different processors. As Tab. 7.1 shows, different block sizes yield optimal performance for different workload and processor combinations. Picking a fixed B ahead of a practical deployment in a dynamic environment will not yield optimal performance. Moreover, even with single single-size workloads it is difficult to predict what block size will be ideal for a concrete processor.

7.2 Specialization is Challenging in Practice

(C1) Developer Complexity. A key barrier to specialization is the increased complexity it introduces for developers. Developers must now reason about which assumptions will reliably hold and can be leveraged to specialize the system for improved performance. Next, the developer must implement the corresponding specialization and evaluate its impact. In practice, this often results

in maintaining multiple code variants. Unsurprisingly, this process is not only complex but also labor-intensive and frustrating, as it frequently becomes an iterative, trial-and-error process.

(C2) Making Optimal Specialization Choices. Optimization decisions are deeply interdependent: the benefit of one specialization often depends on whether others have already been applied. Modern system performance emerges from complex interactions across software layers, hardware behavior, and workload characteristics, making it infeasible for developers to capture these effects using simple local cost models. As a result, determining the optimal specialization strategy statically is fundamentally impossible, since both workload properties and hardware conditions evolve dynamically at runtime. Moreover, optimizations compose nonlinearly: one transformation may expose entirely new optimization opportunities that were previously impossible. For example, PacketMill [127] first de-virtualizes function pointers [223] for the Click modular router [224], and then based on this further eliminates dead code and data structure fields. The analysis for the latter optimization is impossible with the devirtualization as the virtual calls prevent the compiler from analyzing the complete packet handling code.

(C3) Loss of Generality. Specializations may degrade performance or even break the system when the underlying assumptions are violated. For example, a network function may successfully process 99.999% of packets if its processing loop and data structures are specialized in a particular way. However, specializing the system for this dominant class of packets—and thereby simplifying code

and data structures—may break the handling of the rare remaining 0.001% of cases. The need to gracefully handle rare or unexpected cases therefore forces developers to adopt conservative specialization strategies.

(C4) Maintaining Correctness. Dynamic specialization introduces a fundamental correctness and state-management challenge: switching between different specialized implementations at runtime can inadvertently violate program invariants or produce irreversible changes to system state. Specialized versions of code are often generated under different assumptions about control flow, data layout, synchronization behavior, or configuration parameters. As execution transitions between these versions, inconsistencies can arise if state created under one specialization is interpreted differently by another. For example, a specialization may remove fields from a data structure, cache derived values, reorder memory accesses, or elide synchronization operations based on assumptions that no longer hold after a transition.

Chapter 8

Iridescent Design and Implementation

We propose a fundamentally different approach to system specialization for performance and efficiency: *automated online specialization*. We specialize systems at runtime to the exact workload and hardware conditions where the system is deployed, *without the human developer in the loop*. To demonstrate the idea's feasibility, we present a design proposal for a specialization toolbox that can iteratively specialize low-level systems online with runtime performance feedback based on developer-annotated possible *specializations*. We have developed a prototype, Iridescent, and provide evidence for feasibility, performance improvements, and lowered developer effort. We argue that this mechanism of code changes at runtime with full system performance could fundamentally change how we build systems and how we approach compiler optimization.

Specialization	Effect
Value	Convert runtime variables to constants; triggers const. prop., unrolling, vectorization, ...
Assumption	Add boolean assumptions; triggers dead code elimination, branch elimination, ...
Fast-Path	Add if-else based early exits with previously computed/cached results.
Custom Code	Generate custom code at runtime. e.g. dynamic LPM matching via nested branches.

Tab. 8.1: Iridescent specialization types and their effects.

We enable system developers to easily build systems that perform well across a broad set of workloads and platforms with a fundamentally different approach. Instead of exclusively compiling ahead of time, we argue that this requires *online specialization* through code changes as the system runs. This enables one version of the system to automatically adapt to the workload and hardware conditions. Specializing online makes it possible to directly use system performance as feedback for choosing optimal configurations.

8.1 Approach

We enable practical specialization of systems for improved performance and efficiency. Our key idea to enable this given the challenges above, is to *automatically explore a space of possible specializations provided by the developer at runtime based on observed overall system performance.*

8.1.1 Programmatically Specifying the Creative Component

Specifying the Specialization Space. We observe that, while developers often struggle to determine *optimal* specializations and implement systems around them, it is considerably easier for them to identify *possible* points of specialization. We therefore allow developers to annotate system code using simple annotations to define these candidates.

We define a *specialization point* as a location in the program where the behavior of the system can be systematically modified or constrained at runtime (e.g., by fixing a value, inserting alternative code, or enabling a fast path). The collection of all specialization points, together with their possible configurations and values, forms the *specialization space*.

Fig. 8.2b shows an example of such annotations for the matrix multiplication server and illustrates how developers can configure and activate specializations at runtime. In addition, developers may optionally provide LLVM transformations to express more intrusive, application-specific specializations, thereby refining the set of candidate configurations within the overall specialization space. Crucially, Iridescent does not require developers to manually filter or rank these configurations. Tab. 8.1 summarizes the broad classes of specialization points supported by Iridescent and their corresponding effects.

Policy for Exploring Specializations. We rely on developers to provide a policy for exploring the specialization space. Developers control which specialization points to explore and when to explore them, either manually, by leverag-

ing simple search strategies provided by our library, or through existing auto-tuning solutions such as OPPerTune [384]. A key responsibility of this policy is to compare application-specific end-to-end performance metrics, such as request throughput, across different specialization configurations.

8.1.2 Automatically Executing the Mechanical Component

Compilation at runtime. Iridescent uses a JIT compiler at runtime to re-compile specialized versions of the performance critical code. This ensures that the specialized version is thoroughly optimized by the various compiler optimization passes to generate performance efficient code.

Guarded specialization with a fall-back. During JIT-based instantiation of specialized system variants, we treat specializations as transient and dependent on both hardware and workload conditions. To ensure correctness, we inject guard conditions into the generated code that determine whether a specialized version can be safely executed for a given invocation. If the guard condition is satisfied, the execution proceeds along the specialized path; otherwise, the system transparently falls back to the unspecialized version of the code for that invocation. Although evaluating these guards introduces some overhead, they are necessary to preserve correctness. When guard failures are sufficiently rare, the performance gains from executing the specialized fast path in the common case outweigh this overhead, making the specialization beneficial overall.

End-to-end optimality on developer-chosen metrics. We completely forgo cost models and heuristics for predicting which choice in the specialization space

is optimal, and instead explore different points online as the system runs. Our guards ensure that the system always behaves correctly in this process. Since the developer policy uses overall system performance metrics, the metrics also implicitly factor in any overheads, e.g. triggering specialization guards for some calls. This ensures that the system converges to the optimal specialization point for the concrete combination of dynamic system, workload, and hardware conditions.

8.2 Anatomy of an Iridescent System

System Requirements for Iridescent. Iridescent targets performance critical systems implemented in non-managed languages (e.g., natively compiled C/C++/Rust code). We observe that typical performance critical systems code is often architected with an event loop structure – an outer loop handling a sequence of events, such as processing requests or data elements, and an inner handler that processes the event. While Iridescent is designed to specialize these systems in varying dynamic environments, for these systems to fully gain the benefits of Iridescent, these systems should have stability in the workload and operating conditions for a sufficient period of time. Conditions should at least remain stable for the time Iridescent takes to explore multiple different specialized variants where re-compiling a new specialized variant typically takes 10–400 ms, (see §9.4). Based on our experience, conditions should remain stable for at least a minute to get significant performance boosts with Iridescent's specializations.

Isolating performance critical code for specialization. Since Iridescent tar-

```
void matmul(u64 *L, u64 *R, u64 *O, int N, int B) {
  for (int kk = 0; kk < N; kk += B)
    for (int jj = 0; jj < N; jj += B)
      for (int i = 0; i < N; i++)
        for (int jjj = jj; jjj < jj + B; jjj++)
          for (int kkk = kk; kkk < kk + B; kkk++)
            M3[i*N+jjj] += L[i*N+kkk] * R[kkk*N+jjj];
}
int main() {
  while(true) { // Main processing loop
    Req *req = get_req(); Resp *res = new Resp();
    matmul(req.A, req.B, res.C, req.N, 2); }
}
```

(a) Original Code

```
void matmul(u64 *L, u64 *R, u64 *O, int N, int B) {
  B = spec_enum("B",B,2,4,8,16,32,64);
  N = spec_general("N",N);
  ...
}
```

(b) Handler Code

```
Iridescent rt("handler_code.ll");
void spec_policy() {
  Conf N_cs[2] ={{}, {{"N", 256}}};
  do {for (c :cartesian(rt->spec_space(), N_cs)) {
        rt.specialize(c); tput_counter = 0;
        sleep(...); // Sleep for some time to monitor
        if (tput_counter > best) {
          best = tput_counter; best_c = c;}}
    rt.specialize(best_c);
  } while(every 1min);}
int main() {
  auto *matmul = rt.handler("matmul");
  rt.reg_opt_pipeline(opt_passes);
  launch_thread(spec_policy);
  while(true) { // Main processing loop
    Req *req = get_req(); Resp *res = new Resp();
    matmul(req.A, req.B, res.C, req.N, 2);
    tput_counter++; } }
```

(c) Fixed Code

Fig. 8.2: MMulBlockBench Microbenchmark

Iridescent Specialization API	
`spec_enum(lbl, x, ...)`	Specialization space of `x` is the set of enum values
`spec_range(lbl, x, l, h)`	Specialization space of `x` is the range [l,h]
`spec_general(lbl, x)`	Specialization space of `x` is the set of all observed values of `x` observed at runtime
`spec_assume(lbl, cond)`	Adds a boolean condition
`spec_custom_*(lbl, ...)`	Generates custom specialized code
`spec_fp_*(lbl, k, v, N)`	Adds an early-exit fast-path optimization
Iridescent Policy API	
`Iridescent(handler_ir)`	Initialize runtime
`.handler(h)`	Get specialized handler
`.spec_space()`	Obtain code spec. space
`.specialize(c)`	Choose specialization; `c` maps spec. point labels $\mapsto$ values
`.add_custom_spec(n, gen)`	Add app specialization
`.add_custom_pipeline(passes)`	Modify codegen optimizations

Tab. 8.3: Iridescent API overview.

gets performance critical systems implemented in non-managed languages, modifying code of the running system is generally non-trivial. To enable pragmatic specialization, we leverage the event-loop structure commonly found in such systems by requiring developers to separate system code into two components: (i) the *fixed code*, including initialization and the outer loop; (ii) the performance-critical *event handler code*. The developer compiles the fixed code as before, but integrates calls to the Iridescent API (Tab. 8.3). In the handler code, the developer adds lightweight Iridescent specialization API calls. In the fixed code, the developer obtains function pointers for the handlers through the Iridescent policy API, and calls these as before. Fig. 8.2 illustrates this with our matrix multiplication server running example. Fig. 8.2a shows the original code whereas Fig. 8.2c and

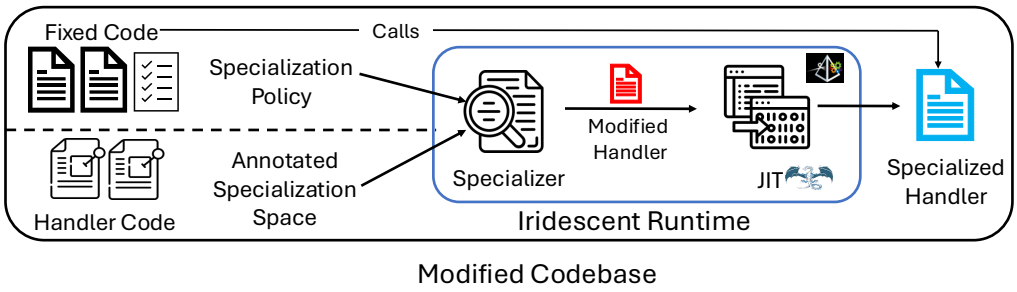


Fig. 8.4: Architecture of an Iridescent-specialized system.

Fig. 8.2b show the decomposed fixed and handler code respectively.

Handling Global State. During dynamic code generation, Iridescent will link handlers against symbols in the fixed code, to enable calls and accesses to global state. Since Iridescent repeatedly rebuilds handlers, *Iridescent requires that the handler code does not include definitions of global variables that must be preserved.* The developer must move all global state to the fixed code, and reference it from the handlers.

Fig. 8.4 shows the overall architecture of an Iridescent-specialized system. The handler code annotates the specialization space and the fixed code provides the exploration policy. Iridescent's runtime specializes the handler based on the exploration policy and optimizes the code using a JIT compiler to produce an optimized and specialized version of the handler. The fixed code then uses the Iridescent API to obtain the address of the specialized handler function and subsequently uses the specialized handler for processing.

8.3 Defining the Specialization Space

The developer defines the specialization space through *specialization point* annotations in the handler code. At runtime, Iridescent replaces these annotations with specialized code, instrumentation, or disables them, guided by the policy. Iridescent offers four types of specialization points: (i) Value Specialization Points; (ii) Assumption Specialization Points; (iii) Fast-Path Specialization Points; and (iv) Custom User-Defined Specialization Points.

8.3.1 Base Specialization Points

Fig. 8.5 shows the usage of the different base specialization points in the handler code as well as the generated specialized code for each of the annotated functions in the handler code. We deliberately choose simple functions to explain the impact of the various specialization points.

Value Specialization Point. A value specialization point signals to Iridescent that the handler code could be specialized to a constant for the wrapped expression. `spec_enum` indicates the value could be one of the specified values; `spec_range` instead specifies a range of integers; `spec_general` leaves it to the policy to determine possible values. For example, in Fig. 8.5, the value of `a` is specialized using the `spec_general` annotation in the function `value_spec_example`. Once the value of `a` is specialized to 4, the specialized function will replace all uses of `a` with the value 4 which would then allow the compiler to do constant propagation that could potentially trigger other optimizations such as loop unrolling and

```
int value_spec_example(int a, int b) {
  a = spec_general("value_a", a);
  return a * b;
}

int assume_spec_example(int a, int b) {
  spec_assume("assume_a", b > 5);
  if (b > 5) {
    return compute_g(a, b);
  } else {
    return compute_h(a, b);
  }
}

int fp_spec_example(int a) {
  int res = compute(a);
  spec_fp("fp_a", a, res, 2);
  return res;
}
```

(a) Annotated handler code

```
int value_spec_example(int a, int b) {
  if (a != 4) throw std::runtime_error("Specialization guard activated");
  return 4 * b; // Further compiler optimizations will switch this to b << 2
}

int assume_spec_example(int a, int b) {
  if (!(b > 5)) throw std::runtime_error("Specialization guard activated");
  return compute_g(a, b);
}

int fp_spec_example(int a) {
  if (a == 4) return 16;
  else if (a == 5) return 25;
  return compute(a);
}
```

(b) Specialized handler code

Fig. 8.5: Demonstration of the Base Specialization Points

auto-vectorization.

Assumption Specialization Point. This provides a *possible specialization assumption* to Iridescent. The condition is a boolean predicate that Iridescent can provide to compiler optimizations to further optimize, through LLVM's builtin `LLVM.assume`. Crucially, the assumption need not always hold. Unlike for the LLVM intrinsic where incorrect behavior results, Iridescent catches this with its specialization guards. For example, in Fig. 8.5, the assumption of `b > 5` in the function `assume_spec_example` is specialized. This assumption then triggers the compiler to do dead-code elimination and remove the specific code branches where the assumption does not hold.

Fast-Path Specialization Point. This specialization point introduces an early-exit optimization that returns cached results when a match is found. It operates in two phases: (i) *instrumentation phase*: Iridescent inserts instrumentation code that samples invocations of the specialized function (at a developer-specified sampling rate) to identify the most frequent inputs along with their computed outputs; and (ii) *specialization phase*: Iridescent rewrites the function into a specialized if-else chain, where the top-N most frequent inputs and their corresponding outputs are materialized as explicit conditional branches to avoid recomputation for hot inputs. The parameter N may be specified by the developer or exposed as a runtime-tunable constant that can itself be further specialized. Inputs that do not match any branch fall back to the original generic implementation. This mechanism is a generalization of hot-key specialization as used in Morpheus [272].

```
class CodeGenPoint {
  public:
    // The generated specialized function
    virtual LLVM::Function *genSpecFn(LLVM::Module &mod) = 0;
    // Generated function that returns the value to be used for specialization checks
    // For most code gen points, specialization checks may be disabled
    virtual LLVM::Function *genSpecCheckValFn(LLVM::Module &mod) = 0;
    // Function that is called for handling the instrumentation call
    // Useful for capturing all the information needed.
    virtual LLVM::Function *getInstrFn(LLVM::Module &mod, LLVM::Function *f) = 0;
    // Resets captured runtime data
    virtual void instrReset() = 0;
    // Sets the code gen type (used iff point has multiple types of generations)
    virtual void setCodeGenMode(uint64 mode, bool is_specialized) = 0;
};
```

Fig. 8.6: Interface for implementing custom code generation specialization points

For example, in Fig. 8.5, a fast-path specialization point is added to the function `fp_spec_example`, specializing it based on the value of input `a` with $N = 2$. The resulting specialized function contains two ordered branches corresponding to the most frequently observed values of `a`, each returning a previously computed result for that input.

8.3.2 Custom User-Defined Specialization Points

Custom code-generation specialization points, annotated using `spec_custom_*`, invoke developer-defined specializations that are registered in the fixed code via `add_custom_spec`. To implement such a specialization point, developers must provide an implementation of the `CodeGenPoint` interface shown in Fig. 8.6.

This interface allows developers to generate custom specialized code in the form of LLVM functions. The most important method in this interface is `genSpecFn`,

which returns an LLVM function containing the generated specialized implementation. In many cases, these code-generation points also require runtime information to construct the specialized code. To support this, developers implement `getInstrFn`, which returns an LLVM function responsible for collecting the necessary runtime data.

An example of a custom code-generation specialization point is the construction of a nested if-else chain as a specialized replacement for an expensive data structure used in Longest Prefix Match computations in low-level network libraries (§9.1.1.2, Fig. 9.1). In this case, the specialization point generates a nested if-else-based matching function derived from the current routing rules. The generated code must be regenerated and re-specialized whenever the routing rules change.

8.3.3 Combining Specialization Points

Multiple specialization points within the same function can be composed to produce further specialized variants of that function. For example, a value specialization point combined with assumption specialization points can enable dead-code elimination. This reduced code can then be further optimized by introducing a fast-path specialization point at the function entry, yielding an even more specialized and efficient version of the function.

Specialization Point Commutativity. However, combining multiple specialization points must be done with care, as their placement is not necessarily commutative for all functions. When specialization points affect overlapping code

regions or interact with the same optimization logic, commutativity may break down because the first applied specialization can change the assumptions, control flow, or intermediate representation seen by subsequent specializations. As a result, applying specialization point A before B can produce a different optimized handler than applying B before A.

For example, placing a value specialization point before a fast-path specialization point may yield a different specialized function than applying the fast-path specialization first. In such cases, the value specialization may enable compiler optimizations such as dead-code elimination, which can remove the condition required for the fast-path specialization, effectively eliminating it from the resulting program.

8.3.4 Specialized Optimization Pipelines

A distinctive capability of Iridescent is that it can specialize not only program code, but also the optimization methodology used to compile that code. Users can further customize compilation pipelines through the `add_custom_pipeline` interface. This pipeline may include both standard LLVM optimization passes and developer-provided custom optimization passes.

Rather than treating the compiler pipeline as static, Iridescent allows optimization strategies (expressed as compiler optimization pipelines) themselves to become specialization points that adapt to runtime-observed behavior. This is particularly useful because compiler optimizations often interact nonlinearly: a transformation that improves performance for one specialization point may

```
/* Context represents the instrumentation and specialization state for the system. */
class SpecCtx {
  public:
    std::map <std::string, SpecPointBase *> points;
    std::set<std::string> trampoline_functions;

    void addPoint(SpecPointBase &point) {
      points[point.label()] = &point;
    }

    SpecPointBase *getPoint(const std::string &name) {
      auto i = points.find(name);
      if (i == points.end())
        return nullptr;
      return i->second;
    }
};
```

Fig. 8.7: Specialization Context that stores the specialization state of the system

degrade performance for another. By specializing the optimization methodology itself, Iridescent can control not only what code is generated, but also how the compiler analyzes and transforms that code, enabling a more adaptive and workload-aware compilation framework.

By default, Iridescent uses the most aggressively optimized out-of-the-box pipeline (e.g., `-O3`) for the entire duration of system execution.

8.3.5 Tracking Specialization State

Iridescent's runtime may choose to activate different specialization points at different periods of execution. Consequently, Iridescent needs to track the specialization state of the system. As part of this state, Iridescent captures all the defined specialization points, their current state as to whether that point is active

or not, the specific value (if any) to which each specialization point is specialized, and any additional statistics associated with each point.

To capture this state, Iridescent maintains a *specialization context*, shown in Fig. 8.7. The Iridescent runtime maintains a single instance of the `SpecCtx` class, which encapsulates the entire specialization state of the system. The context stores a list of all specialization points in the system, where each specialization point corresponds to an object of type `SpecPointBase`, the abstract base class used to implement specialization points. Each specialization-point object maintains its own state, including the set of possible specialization values, information collected during runtime execution, and the current activation state of the specialization. This state can then be used to make specialization decisions either automatically within the runtime or explicitly through API calls from the fixed code.

8.4 Defining Specialization Policies

The developer implements the specialization policy in the system's fixed code using the Iridescent policy API. The policy determines what to specialize, when to specialize, and how specialization should be performed.

Fig. 8.2c illustrates how developers can control the exploration and selection of specializations. In line 4, the `spec_space` API returns the specialization space generated from the annotations in the handler code. We combine this space with other available specialization choices using a Cartesian product to construct the full set of candidate configurations. In lines 4–8, the fixed code

```
class ExplorationStrategy {
  public:
    virtual Configuration next(SpecCtx &ctx) = 0;
    virtual std::string name() = 0;
    virtual Configuration getNextConfig() { ... }
    virtual void set_score(score uint64) { ... }
    virtual Configuration getBestConfig() { ... }
};
```

Fig. 8.8: Interface for implementing custom exploration strategies

evaluates these configurations and selects the best-performing one. To make this selection, the specialization policy uses an end-to-end performance metric, in this case throughput. The fixed code triggers re-exploration at fixed time intervals to adapt to changes in workload.

Developers may also enable instrumentation for selected specialization points to dynamically identify promising opportunities for specialization. For example, a policy for the matrix multiplication server may detect the most frequent matrix size N and only enable specialization if at least 70% of requests use that value.

8.4.1 Exploring the Specialization Space

Exploration Engine. Iridescent provides an exploration engine that automatically evaluates different specialization *configurations*—that is, combinations of active specialization points and their assigned specialized values—and measures the performance of the resulting system in order to identify the best-performing configuration. To support this process, Iridescent includes a periodic exhaustive search strategy for simpler cases, which can be used to explore the system's specialization space in a straightforward manner.

Custom Exploration Strategies. Iridescent provides a lightweight interface, shown in Fig. 8.8, for implementing custom search strategies. To define a new exploration strategy, developers only need to implement two functions: `next` and `name`. The `name` function returns the identifier of the strategy, while the `next` function returns the next specialization configuration to be evaluated by the exploration engine. The base abstract class `ExplorationStrategy` provides the necessary infrastructure to integrate custom strategies with Iridescent's exploration engine. This includes logic for tracking the best-performing configuration and selecting the final configuration for deployment. We expect that most systems will either use existing auto-tuning strategies or implement custom strategies tailored to maximize end-to-end system performance using these Iridescent building blocks.

Large-Scale Exploration. For more complex scenarios where the specialization space is too large for exhaustive search, Iridescent integrates with OPPerTune [384] to leverage its state-of-the-art configuration search strategy. The OPPerTune-based exploration strategy implements the interface shown in Fig. 8.8. Currently, OPPerTune runs as a separate gRPC service, and Iridescent's exploration engine communicates with it remotely to request new specialization configurations and report observed performance metrics. To enable correct integration with OPPerTune, Iridescent encodes each specialization point as a categorical tuning knob and normalizes configuration scores to the range $[0, 1]$.

8.4.2 End-to-End Metrics as Optimization Targets

```cpp
struct Stats {
public:
   // Map to store the collected statistics as key-value pairs
   std::unordered_map <std::string, uint64_t> stats;
};

/** Objective function type definition that returns a number
 * based on measured stats. The number signifies how good
 * the measured statistics are. Higher the number, better
 * the configuration that led to it.
 **/
typedef std::function<uint64_t(Stats&)> ObjectiveFn;

/** MeasurementFn function type definition that returns
 * the measured stats.
 **/
typedef std::function<Stats(void)> MeasurementFn;
```

Fig. 8.9: Iridescent support for supporting end-to-end metrics as optimization targets

To identify the best specialization, Iridescent supports optimization using end-to-end metrics. Since different systems may optimize for different goals, Iridescent does not hard-code or assume any specific metric by default. Instead, developers define the relevant end-to-end metrics as part of the specialization policy.

To support user-defined metrics within Iridescent's exploration engine, developers implement two types of functions, shown in Fig. 8.9, which are directly integrated into the exploration loop. The first is a *measurement function*, which collects all relevant system metrics and stores them as key-value pairs. The second is an *objective function*, which consumes these metrics and computes a scalar score representing configuration quality. By default, the exploration engine selects the specialization configuration with the highest score.

During exploration, the engine first requests a new specialization configuration from the exploration strategy. It then applies this configuration using the specialization runtime and runs the system for a configurable observation period. After this period, it invokes the measurement function to collect end-to-end metrics and subsequently calls the objective function to compute the corresponding score for the observed configuration.

8.5 Specialization Runtime

The specialization runtime consists of two key components: (i) the *specializer*, which generates code by applying selected specializations; and (ii) the *JIT compiler*, which compiles and optimizes the generated specialized code and makes it available to the fixed code.

8.5.1 Specializer

The specializer generates the specialized version of the handler code. It operates at function granularity, producing function variants in which specialization points are replaced according to the chosen specialization configuration.

For each specialization point, Iridescent performs a set of transformations based on the configuration supplied by the policy. If the policy marks a specialization point as inactive, the Iridescent specializer removes the annotation and restores the original value, or skips the point entirely if it is an assumption point. For an active value specialization point, Iridescent replaces the specialization annotation in the handler code with the constant value chosen by the explorer. By

default, the specializer also inserts a specialization guard, although developers may explicitly disable this behavior. For a custom specialization point, the specializer replaces the specialization annotation with custom source code generated by the handler previously registered for that specialization-point type. Once specialization is complete, the recompilation process enables standard compiler optimizations to exploit the newly introduced constants or generated code.

Instrumentation. Some specialization policies benefit from collecting runtime data to generate candidate specialization configurations. To support this, the specializer can optionally enable instrumentation for individual specialization points. When instrumentation is enabled for a specialization point, Iridescent generates additional code to collect and record the observed runtime values associated with that point. The policy can then retrieve this information through the result of the `spec_space` call. By default, the specializer injects data collection code at the same location where the specialized code is to be inserted. However, some specializations require collecting data from different parts of the code. To support this, the specializer allows developers to specify auxiliary data collection annotations in the handler code. The auxiliary data collection annotations re-use the same label as the original specialization point declaration to ensure the data is linked to the specialization points correctly. During the instrumentation phase, the specializer replaces these auxiliary annotations with the necessary code to collect the relevant runtime data.

8.5.2 Online Compilation with JIT

```
void __cleanup_foo() { // User-defined function that restores state }

int foo_specialized(int a, int b, int c) {
 if (a != 5) throw std::runtime_error("Specialization Guard activated");
 /* Specialized Code */
 ...
}

int foo_original(int a, int b, int c) { /* Unspecialized Code */ ... }

int foo(int a, int b, int c) {
 try {
  return foo_specialized(a, b, c);
 } catch(std::runtime_error) {
  __cleanup_foo();
  return foo_original(a, b, c);
 }
}
```

Fig. 8.10: Contents in JIT for a specialized function

Compiling the specialized code. The specialization runtime adds the specialized code generated by the specializer to the JIT compiler. The JIT then compiles the code and executes a developer-specified pipeline of optimization passes, including any custom optimization passes provided by the developer, to generate an optimized version of the specialized code. By default, the JIT applies the standard `-O3` optimization pipeline when the developer does not specify a custom optimization pipeline.

JIT contents. Fig. 8.10 shows the contents of the JIT for an example specialized function. For each specialized function, the JIT maintains three variants of the function. The first variant is the specialized function (`foo_specialized`), which contains the generated specialized code. The second variant is the origi-

nal, non-specialized implementation (`foo_original`). This generic version serves as a fallback when the specialized code is not applicable. The third variant is the trampoline function (`foo`), which orchestrates execution between the specialized and original handlers. In addition, the trampoline catches exceptions raised by specialization guards and performs any necessary cleanup or state restoration for modifications made by the specialized handler before the guard failure occurred.

Using the specialized code. To invoke specialized handlers, the fixed code obtains function pointers to the trampoline functions from the specialization runtime. The fixed code then uses these pointers to execute the specialized handlers. The trampoline function resides at a fixed address and does not change across specialization iterations. Consequently, the fixed code only needs to retrieve the function pointers once at the start of execution. Currently, developers must explicitly use the Iridescent API to obtain the address of the trampoline function at runtime.

8.5.3 Correctness through Specialization Guards

Iridescent inserts a runtime check, called a *specialization guard*, for each enabled specialization point. Policies may selectively disable these guards for individual specialization points. The specialization guard ensures that the assumptions under which the specialized code was generated continue to hold during execution. To enforce this, the specializer inserts an early exit from the specialized code path when a guard fails by throwing an exception. The Iridescent trampoline function catches these exceptions and transparently redirects execution to the

original non-specialized version of the function without exposing the exception to the fixed code.

These guards are necessary because the specialized handler code following the guard is compiled and optimized under the assumption that the guard condition always holds. Without an explicit guard check and fallback mechanism, the specialized code could execute even when its assumptions are violated, potentially resulting in incorrect or undefined behavior.

This specialization guard mechanism allows developers to benefit from aggressive code specialization while preserving correctness and generality.

Restoring state and side effects. To ensure correct restoration of the state, the specializer will call a developer-defined cleanup function for reversing any side-effects before throwing the exception. It is critical to note here that not all side-effects are reversible (*e.g.* sending a packet to a neighbor), so developers should choose specialization points carefully.

8.6 Prototype Implementation

We have implemented Iridescent in $5.5K$ lines of C++. Our implementation uses the LLVM IR and JIT [235] to generate the specialized code at runtime. Currently, Iridescent's prototype provides first-class support for systems implemented in C and C++. However, Iridescent's specializer operates on the LLVM IR which makes it easy to extend Iridescent's specialization to other languages, such as Rust, that have LLVM support.

8.6.1 Specializer Implementation

As part of the specialization, Iridescent's specializer parses the LLVM IR of the annotated handler code to generate the specialized, original, and trampoline functions for each function. The specializer is implemented as a set of LLVM transformation passes that operate on the LLVM IR of the instrumented handler code. The sequence of the custom passes is as follows:

1. `EntryPass`: This is a LLVM Analysis pass that extracts the names of all the functions that contain annotations for specialization points. These functions are treated as entry points into the JIT and are essentially the top-level functions in the handler code.

2. `CleanupPass`: This is a LLVM Analysis pass that extracts the mapping of all the cleanup functions to the entry functions, if provided.

3. `OriginalPass`: This is a LLVM Transformation pass that creates a copy of each entry function and adds a `_original` suffix to the name of every duplicated function. This pass also deletes the user-defined annotations.

4. `SpecializationPass`: This is a LLVM Transformation pass that creates the specialized version of each entry function. If this is the first time this pass is being executed at runtime, then this pass also creates the specialization points corresponding to the annotations and adds them to the specialization context (Fig. 8.7). Future executions of this pass refer to the specialization context for deciding the specific specialization to apply.

5. `InlinerPass`: This is a LLVM Transformation pass that marks all non-entry functions to be inlined to enable additional compiler optimizaitons.

6. `Custom Passes`: These are the user-defined custom LLVM passes for application-specific optimization.

7. `TrampolinePass`: This is a LLVM Transformation pass that generates the trampoline functions for each specialized function. The name of the trampoline function matches the name of the entry function extracted in the `EntryPass`.

After these passes finish execution, Iridescent's specializer runs the registered pipeline of compiler optimization passes and adds the resultant output to the JIT. The execution of these passes happen periodically as specified by the specialization policy.

Chapter 9

Evaluation

In this section, we evaluate how much Iridescent improves performance of systems by enabling online specializations. We show how we can use online specialization with Iridescent for three use-cases: (i) enabling compile-time optimizations at runtime; (ii) enabling incremental specializations at runtime; (iii) design exploration at runtime. We answer the following questions:

- Do Iridescent-enabled optimizations improve performance? (§9.2)
- Can Iridescent find an optimal specialization through exploration at runtime under dynamic conditions? (§9.3)
- What is the overhead of using Iridescent? (§9.4)

9.1 Experimental Setup

9.1.1 Case Studies

We evaluate Iridescent with four open-source systems and libraries and the MMulBlockBench microbenchmark. We explain each system along with the changes we made for integrating Iridescent below.

9.1.1.1 Tiling Benchmarks

MMulBlockBench. We implement a blocked (tiled) version of the matrix multiplication algorithm [70, 430, 431]. To integrate this benchmark with Iridescent, we convert the tile size, B, into a value specialization point.

CloverLeaf. We use a C implementation of the CloverLeaf application from the SPEC ACCEL2023 benchmark [89]. In this application, we implement tiled versions of various mathematical kernels and introduce value specialization points for the tile sizes, T_i and T_j. In total, the application contains 24 unique specialization points, each of which can take 7 distinct values, resulting in a specialization space of 7^{24} possible configurations.

9.1.1.2 LibLPM

LibLPM [334] is an open-source Longest Prefix Match (LPM) library written in C with built-in support for IPv4 and IPv6 addresses. LPM is a fundamental operation in packet routing for finding the closest matching route in routing tables for incoming packets.

To integrate LibLPM with Iridescent, we implement three different special-

```
int lpm_lookup_ifelse(lpm_t *lpm, const void *addr, size_t len) {
   uint32_t addr_value = (uint32_t) addr;
   if ((addr_value & ((1 << 8) - 1)) == 0xBE) {
      if ((addr_value & ((1<<16)-1)) == 0x14BE) {
         if ((addr_value & ((1<<24)-1)) == 0x14BE) {
            if ((addr_value & ((1<<26)-1)) == 0x14BE) {
               if ((addr_value & ((1<<28)-1)) == 0x14BE) {
                  return 28;
               }
               return 26;
            }
            return 24;
         }
         return 16;
      }
      return 8;
   }
  if ((addr_value & ((1 << 8) -1)) == 0xCE) {
    return 5;
  }
    return 0;
}
```

Fig. 9.1: Sample LibLPM-NI generated code

izations:

LibLPM-FP specializes the lookup function to add a fast-path specialization point. This specialization adds an exact match if-else fast-path for the top-N frequently observed input IP addresses.

LibLPM-NI introduces a code-generation specialization point into the LPM library. This specialization generates a new version of the lookup function containing a nested if-else tree of prefix-match checks for incoming addresses, with one branch corresponding to each LPM entry. The generated tree begins with the least specific matching rule (based on prefix length) and progressively checks increasingly specific rules until no more specific match is found. By embedding prefix rules directly into the generated code, this specialization eliminates expensive dynamic data-structure traversals and exposes additional opportunities for branch-level compiler optimizations. Fig. 9.1 shows an example of LibLPM-NI generated code for an LPM table containing six routes.

LibLPM-NI-FP combines the previous two approaches by adding a fast-path specialization point to the nested-if specialized lookup function generated by LibLPM-NI.

9.1.1.3 TAS

TAS [220], or TCP Acceleration as a Service, is a lightweight software TCP fast path optimized for common-case client–server RPC workloads. TAS executes common-case TCP operations in an isolated fast path built on DPDK [113], while delegating corner cases to a slower execution path.

To integrate TAS with Iridescent, we convert the `BATCH_SIZE` macro into a value specialization point. In TAS, `BATCH_SIZE` is used in three different contexts: (i) determining the number of packets read from the NIC, (ii) determining the number of packets read from the application queues, and (iii) determining the number of packets read from the queue manager. To enable more fine-grained control, we transform each of these uses into separate runtime constant specialization points: (i) `rx_batch`, (ii) `queues_batch`, and (iii) `qman_batch`.

9.1.1.4 FastClick

FastClick [45] is an extended version of the Click Modular Router [224] that adds Netmap and DPDK support for running the modular router in user space. A FastClick (or Click) router is constructed from individual packet-processing modules called *elements*. Each element implements a simple router function, such as packet classification, scheduling, routing, or interaction with network devices. Users configure routers by composing graphs of connected elements.

To integrate FastClick with Iridescent, we modify the `LinearIPLookup` element to create an Iridescent-enabled version. The original element uses a linear search algorithm to identify the best matching route for an incoming packet based on the routing table. We extend the packet-processing function of `LinearIPLookup` by introducing a fast-path specialization point. The target of this specialization point is the element's internal lookup function.

9.1.1.5 Network Functions

Network Functions is a suite of the network functions that include DPDK and eBPF implementations of common network functions such as NATs, Routers, and policers, among others. These functions have been developed and used for evaluation in various research projects, including Pix [199] and Vigor [450]. We extract the implementations from the Pix artifact [198].

To integrate the Network Functions with Iridescent, we convert the `BATCH_SIZE` used by the DPDK version of these functions into a runtime constant specialization point. In the network functions, the BATCH_SIZE controls the number of packets that are read from a network port at a given time.

9.1.2 Experimental Testbed

For our experiments, unless otherwise stated, we configure two identical machines as the client and server. They are directly connected with a pair of 100 Gbps Mellanox ConnectX-5 Ethernet adapters. Each machine has two Intel Xeon Gold 6152 processors at 2.1 GHz, each with 22 cores for a total of 44 cores and 187 GB of RAM per machine. Each machine runs Linux kernel 5.15 with Debian 11.

9.2 Compiler Optimizations at Runtime

Compile-time optimizations often produce significantly more efficient code. These optimizations include constant propagation, loop unrolling, dead-code elimination, and the use of vector instructions. With Iridescent, we can enable such compile-time optimizations dynamically using runtime information.

Machine (Processor)	**Constant** (c) (cycles/op)	**Variable** (v) (cycles/op)	**Benefit** (v/c)
IceLake	175297	284944	61%
IvyBridge	250434	661295	246%
CoffeeLake	168817	581130	348%
AlderLake-p	173350	583724	336%
AlderLake-e	206924	557572	269%

Tab. 9.2: Impact of turning block size, B, as compile-time constant at runtime for N=64 for different hardwares (Tab. 7.1)

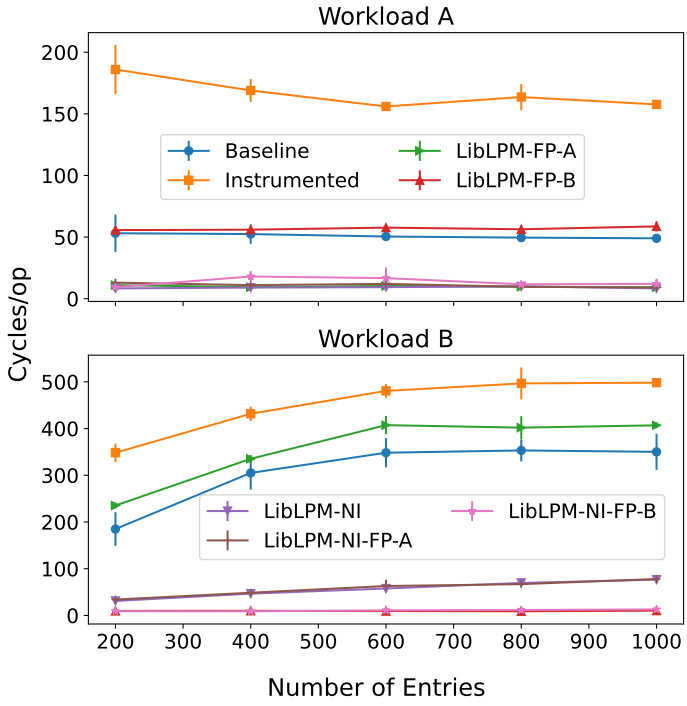


Fig. 9.3: Average cycles required to perform a table lookup with varying table sizes under two different workloads.

Iridescent enables cascading compiler optimizations. We first demonstrate the benefits of enabling compiler optimizations at runtime using the MMulBlock-Bench microbenchmark shown in Tab. 7.1. Specifically, for the workload $N = 64$, we compare two configurations: one in which the optimal block size B remains a runtime parameter, and another in which Iridescent converts B into a compile-time constant at runtime, thereby enabling downstream compiler optimizations. For both configurations, we execute the function a fixed number of times and measure the average number of cycles per function execution. Tab. 9.2 shows the benefit of converting B into a compile-time constant across different machines. For every machine, specialization reduces execution cost by at least 50%, and for four of the five operating conditions, the reduction exceeds 240%. These improvements arise because Iridescent's runtime specialization enables multiple compiler optimizations to cascade and reinforce one another, ultimately producing a more efficient version of the code. In this case, specializing B allows the JIT compiler to unroll the inner loops in the matrix multiplication implementation and subsequently leverage optimized vector instructions.

Iridescent incorporates existing specializations. To demonstrate Iridescent's ability to incorporate existing runtime specialization techniques, we implement Morpheus' [272] map hot-key specialization as a fast-path specialization point within the packet processing function of FastClick's LinearIPLookup element. We configure this fast-path specialization point with a 0.01% sampling rate during the instrumentation phase. We execute the FastPath router on one machine, treating it as the device under test, while using Pktgen [315] to generate an open-

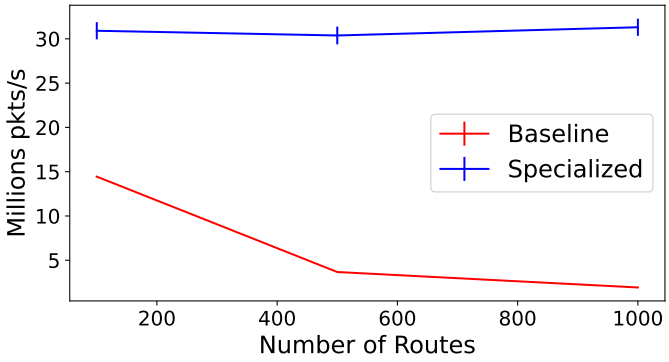


Fig. 9.4: FastPath router throughput with Iridescent specialized LinearIPLookup element for different routing table sizes at 100% fast-path hit rate.

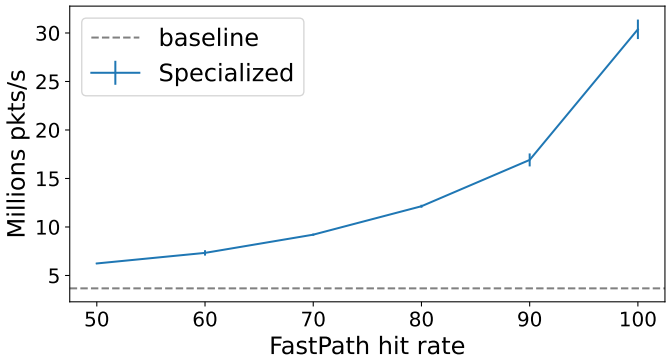


Fig. 9.5: FastPath router throughput with Iridescent specialized LinearIPLookup element for varying fast-path hit rates.

loop workload consisting of packets with destination IP addresses drawn from a configured set. We repeat the experiment across different routing table sizes in LinearIPLookup as well as different fast-path sizes in order to evaluate throughput under varying conditions. Fig. 9.4 shows that Iridescent achieves up to a 15× improvement in throughput, depending on routing table size, under a 100% fast-path hit rate. The improvement increases with larger tables, since the baseline must perform longer linear scans. Fig. 9.5 shows that for a router with a 1000-entry routing table, even at a 50% fast-path hit rate, Iridescent achieves approximately a 5× throughput improvement. The performance gains further increase as the fast-path hit rate increases.

Iridescent enables custom specializations. Next, we demonstrate the benefits of custom compile-time optimizations for LibLPM. For this experiment, we evaluate eleven different configurations arising from a combination of five LPM table sizes and two workloads: Workload A and Workload B. In Workload A, incoming IP addresses match a specific, high-selectivity prefix entry in the routing table. In Workload B, incoming IP addresses match only the default LPM entry with prefix length 0. Thus, Workload A and Workload B represent the best-case and worst-case scenarios, respectively, for the LPM lookup function. We execute these workloads with seven specializations:

(i) Baseline: no Iridescent-enabled specialization.

(ii) Instrumented: Iridescent-enabled specialization capturing the most popular inputs (10% sampling); data used to guide the fast path specializations.

(iii) LibLPM-FP-A: fast-path specialization of size 1 specialized for Workload A.

(iv) LibLPM-FP-B: fast-path specialization of size 1 specialized for Workload B.

(v) LibLPM-NI: nested-if code generation specialization.

(vi) LibLPM-NI-FP-A: Combine LibLPM-NI and -FP-A.

(vii) LibLPM-NI-FP-B: Combine LibLPM-NI and -FP-B.

Fig. 9.3 shows the average number of cycles per LPM lookup lookup function for all different combinations of configurations and specializations. For Workload A, all three specializations, LibLPM-FP-A, LibLPM-NI, LibLPM-NI-FP-A, achieve up to $6\times$ reduction in cycles per execution. However, unlike the LibLPM-FP and LibLPM-NI-FP-A specialization, this LibLPM-NI specialization does not require an additional instrumentation phase to sample executions to find the optimal candidates for the fast path. LibLPM-NI improves performance across the board for all workloads by up to $6\times$. However, despite this improvement, LibLPM-NI is not the best-performing specialization for Workload B. In this case, the *-FP-B specializations perform best, achieving up to a $30\times$ improvement, as nearly all executions of Workload B take the fast path.

9.3 Runtime Design Exploration

Iridescent enables runtime design exploration. We evaluate runtime design exploration using TAS [220]. The experiment is conducted in a standard server-client setup, where the server provides an echo service and the client generates an open-loop workload of 64-byte packets using multiple threads. Both the server and client are TAS-enabled. In this experiment, we modify the server-side TAS instance and measure throughput as millions of packets processed per

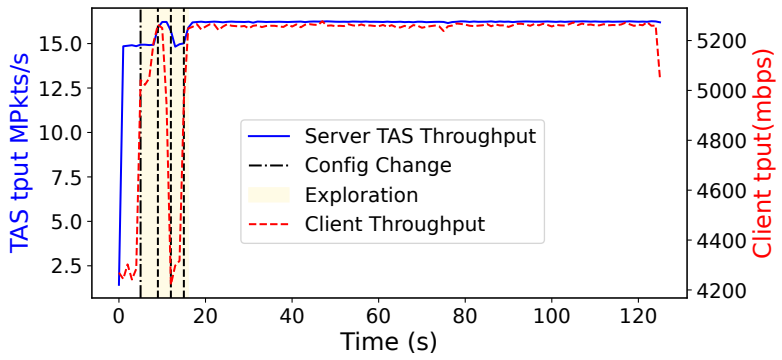


Fig. 9.6: Automatic exploration and specialization of different configurations for TAS.

second. With Iridescent, we explore different values of the `rx_batch` parameter in the server-side TAS source code. Fig. 9.6 shows that Iridescent automatically identifies and selects the best-performing configuration by exploring different batch size values across relevant code locations.

Iridescent automatically adapts to changing workloads. To demonstrate that Iridescent can adapt automatically to changing workloads, we first use the MMulBlockBench microbenchmark. We compare a Iridescent-enabled version of the function against a non-Iridescent baseline. In the baseline, the block size B is exposed as a runtime parameter. In the Iridescent-enabled version, Iridescent explores different values of B by specializing it in the generated code, effectively converting B into a compile-time constant for each candidate configuration.

We execute both versions under two distinct workloads, each lasting one minute and presented sequentially. We measure overall throughput (execution-s/s) for both configurations. Fig. 9.7 shows how Iridescent explores different block sizes and identifies the most performant configuration compared to the

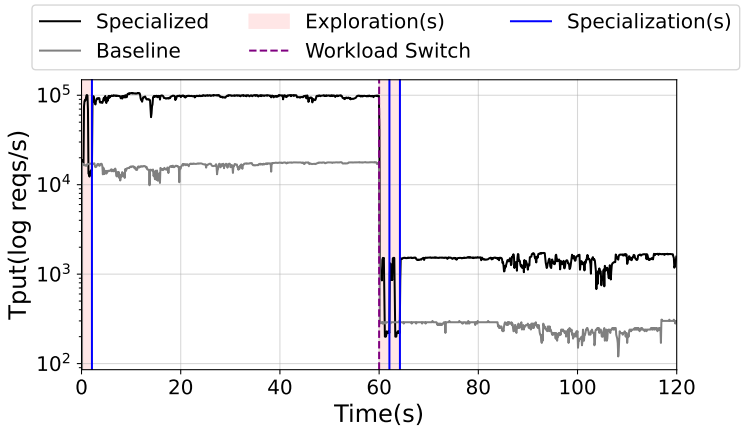


Fig. 9.7: Automatic exploration and specialization of MMulBlockBench for two different workloads.

baseline. Moreover, Iridescent automatically detects the workload change via a drop in throughput and re-initiates the A/B testing process to adapt to the new conditions.

Next, we evaluate runtime design exploration and adaptation for two network functions, Vignat and Vigpol, from Vigor [450]. The experimental setup consists of a packet generator running on one machine and a device under test (DUT) on another machine. The DUT exposes two ports: one serving as the internal-facing port and the other as the external-facing port. For each incoming packet generated by the traffic generator, the DUT executes the corresponding network function. The implementation of the network functions includes different execution paths depending on whether packets arrive on the internal or external port. The network functions are configured with an Iridescent specialization policy that triggers an exploration phase to determine the best-performing `BATCH_SIZE` whenever a change in workload is detected. The experi-

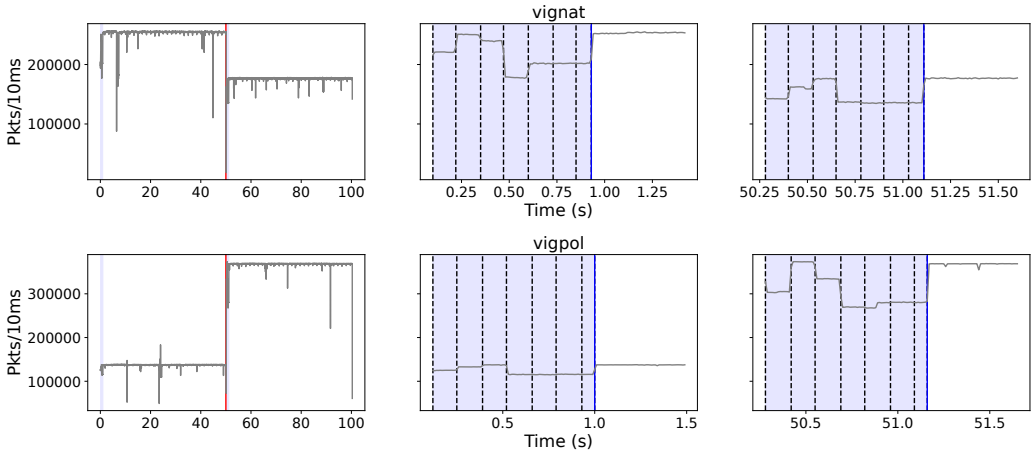


Fig. 9.8: Batch size exploration with Network Functions

ment is executed in two phases: in the first phase, packets arrive exclusively on the external-facing port, while in the second phase, packets arrive exclusively on the internal-facing port. Fig. 9.8 shows the results for both the NAT (Vignat) and Policer (Vigpol) network functions. In each phase, Iridescent successfully identifies the best-performing configuration, with different optimal `BATCH_SIZE` values emerging under the two distinct workload conditions.

Iridescent transparently enables exploration-based adaptation with custom specializations. We evaluate runtime design exploration for FastClick's LinearIPLookup. The experimental setup consists of a FastPath router running on one machine, which serves as the Device Under Test, and a separate traffic generator (Pktgen) that produces an open-loop workload. Pktgen sends packets with destination IP addresses drawn from a fixed set of addresses, which defines the initial workload. At the 1-minute mark, we introduce a workload shift by switching to a completely different set of destination IP addresses with no over-

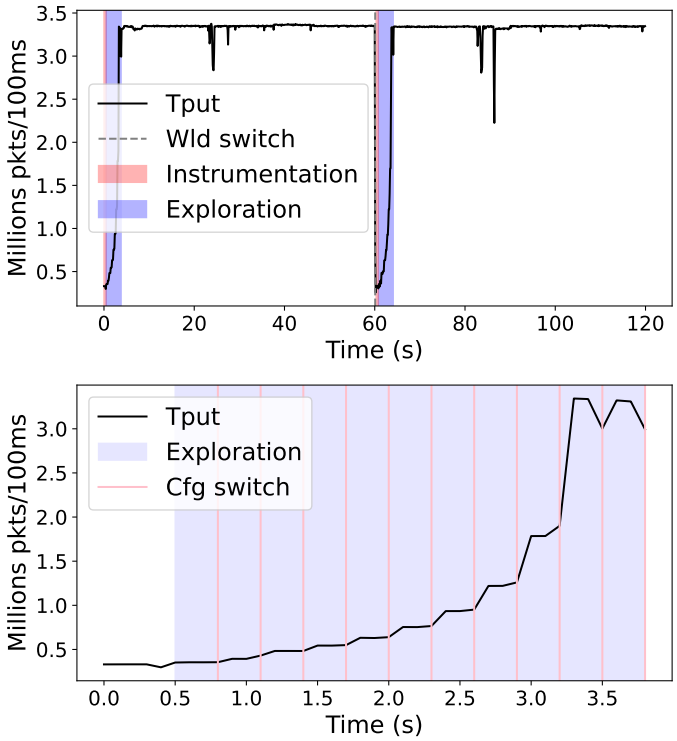


Fig. 9.9: Optimal fast path size exploration with Iridescent.

lap with the initial set. The router is configured with an Iridescent specialization policy that triggers exploration when it detects a significant change in measured throughput (at least 25%). Fig. 9.9 shows the resulting throughput over time for the two workload phases. At startup, the policy triggers exploration due to the sudden influx of traffic, transitioning from zero incoming packets to a sustained load.

The adaptation process proceeds in stages. First, Iridescent performs a short instrumentation phase (approximately 100 ms) to identify the most frequently observed destination IP addresses. It then transitions into an exploration phase, where it evaluates different fast-path configurations, specifically varying the number of IP addresses encoded directly into the generated conditional (if-else) chain. This determines the size of the specialized fast-path code. Once exploration completes, Iridescent selects the best-performing configuration and generates specialized code, which remains active until the workload change at the 1-minute mark. After the workload switch, the specialization policy again detects a significant change in throughput and automatically re-triggers the instrumentation and exploration phases. As shown in Fig. 9.9, Iridescent is able to rapidly adapt to workload shifts and continuously regenerate optimized specialized code under dynamic execution conditions.

Iridescent can explore large specialization spaces. We evaluate Iridescent's ability to navigate large specialization spaces using the CloverLeaf benchmark, coupled with OPPerTune's exploration algorithm [384]. OPPerTune is a measurement driven auto-tuning method that iteratively evaluates candidate config-

urations and guides search toward high-performing regions of the configuration space based on observed runtime performance.

In this experiment, the full specialization space consists of approximately 7^{24} possible configurations, arising from multiple tunable parameters in the CloverLeaf application. To keep exploration tractable, we restrict the search budget to 20 configurations sampled and guided by OPPerTune's strategy. Despite this severely limited exploration budget relative to the full space, Iridescent is able to identify a configuration that reduces the number of cycles per iteration of the main CloverLeaf loop by approximately $3\times$ on average.

These results demonstrate that, by leveraging state-of-the-art auto-tuning exploration strategies, Iridescent can efficiently identify high-quality configurations even within extremely large specialization spaces.

9.4 Overhead of Iridescent's Abstractions

Compilation cost. Tab. 9.10 shows the JIT compilation time for each of the target systems. Note that this compilation occurs off the critical path and therefore does not constitute a performance bottleneck. However, this cost does affect the propagation delay, *i.e.*, the time required for the specialized version of the code to become available for execution.

To analyze how compilation time changes with increasing JIT code size, we consider the code generated by Iridescent for the LibLPM-NI specialization. Fig. 9.11 shows the compilation time for the LibLPM-NI specialization as a function of the number of elements in the LPM lookup table. The compilation time increases lin-

System	Time (ms)	Size (KB)
MMulBlockBench	10 ± 1	10
LibLPM-FP	72 ± 9	80
Network Functions	98 ± 5	67
TAS	340 ± 5	360
FastClick	11 ± 1	9.2

Tab. 9.10: JIT compilation time and corresponding size

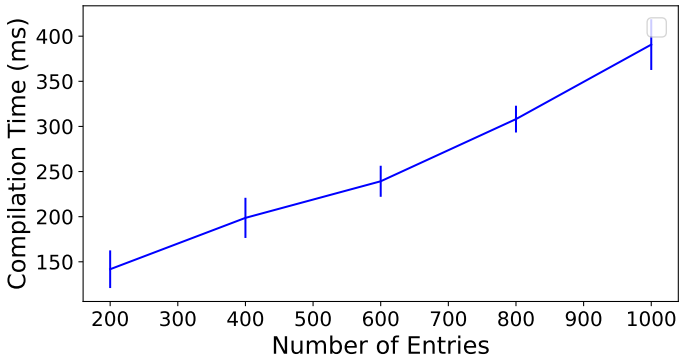


Fig. 9.11: Compilation time of LibLPM-NI (nested-if specialization) as a function of the number of elements in the LPM lookup table.

early with the number of LPM entries. This occurs because Iridescent generates one basic block in the specialized code for each LPM entry. Consequently, the size of the generated code also grows linearly with the total number of entries.

Developer cost. Tab. 9.12 shows the necessary lines of code changes required to integrate Iridescent specializations into the target systems. In all cases, the integration required fewer than 100 lines of code changes. Implementing the custom LPM specialization required only 162 lines of code to enable the user-defined LPM specialization.

Instrumentation overhead. To measure the overhead of instrumentation, we

System	Handler Annotations (LoC)	Spec Policy (LoC)
MMulBlockBench	2	47
LibLPM	4	9
Network Functions	5	43
TAS	8	38
FastClick	5	88

Tab. 9.12: Lines of Code change required to integrate Iridescent.

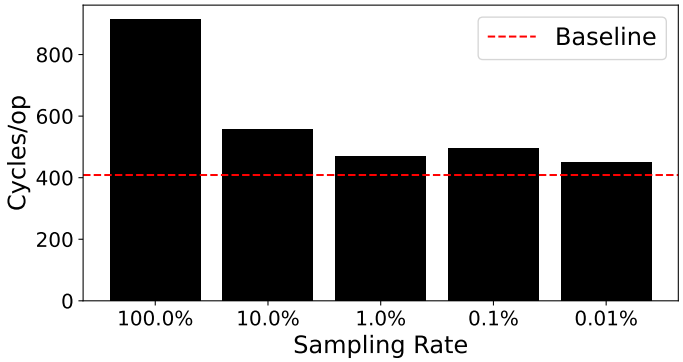


Fig. 9.13: Impact of sampling rate on instrumentation overhead for LibLPM-FP configuration for Workload B.

devise a microbenchmark called SimpleBench consisting of two simple functions, `f` and `g`. Function `f` computes the square of its input, while `g` computes the product of its two inputs. For `f`, we add a general specialization point for its input `a`. For `g`, we add a range-based specialization point for its second input `b`. For both specialization points, `a` and `b`, we first execute their respective functions in normal mode and then use Iridescent to add instrumentation. Each configuration is executed one million times. As a baseline, each function requires approximately 8 cycles per execution in normal mode. For `a`, which uses a general specialization point, instrumentation introduces an overhead of approximately 450–500 cycles per execution. In contrast, for `b`, which uses a specialization point designed for efficient instrumentation of values within fixed ranges, instrumentation adds only around one extra cycle per execution.

Whenever possible, Iridescent users should therefore prefer the more efficient specialization-point design. However, this is not always feasible, and users may need to rely on the more general specialization point. To mitigate the resulting overhead in low-latency computations, Iridescent allows users to specify a sampling rate for instrumentation.

Fig. 9.13 illustrates how reducing the sampling rate for the LibLPM-FP Workload B configuration decreases instrumentation overhead. In this experiment, we execute the instrumentation-enabled LibLPM-FP implementation for Workload B one million times under different sampling rates. As the sampling rate decreases, the amortized instrumentation overhead correspondingly drops. At sampling rates below or equal to 1%, the overhead becomes negligible, and nor-

mal function execution dominates overall runtime cost.

Specialization guards and failures. To measure the cost of specialization guards and failures, we use the SimpleBench microbenchmark. For function `f`, we use Iridescent to specialize the value of `a` and generate two different versions of the function. In the first version, we disable the specialization check at the entry point of `f`. We then execute this specialized version one million times using the specialized value as input. Each execution takes 7 cycles. In the second version, we enable the specialization check at the entry point of `f`. If the specialization check fails, *i.e.* the input value of `a` does not match the specialized value, this version of `f` throws an exception that is caught by Iridescent-inserted trampoline code, which subsequently invokes the generic version of the function. We first execute this version with inputs such that the specialization guards always pass. The specialization check adds approximately one extra cycle per execution. Next, we execute the same version with inputs such that the specialization guards always fail. In this case, exception handling and fallback execution introduce approximately 5000 cycles of overhead per invocation. Thus, the raw cost of a specialization check is approximately one additional cycle, whereas the cost of a specialization failure is approximately 5000 cycles.

9.5 Summary of Results

Our evaluation demonstrates that Iridescent-enabled optimizations can significantly improve application performance across diverse workloads and deployment conditions. Through runtime exploration, Iridescent is able to effectively

identify and converge on high-performing specializations even under dynamic and changing system conditions, enabling the system to adapt to variations in workload characteristics and execution environments. Furthermore, the results show that the overhead introduced by Iridescent is low relative to the performance gains achieved, indicating that the benefits of runtime specialization and exploration outweigh the associated costs. Collectively, these findings show that Iridescent provides an effective and practical approach for adaptive runtime specialization in complex cloud systems

Chapter 10

Related Work

10.1 Offline Automatic Specialization

Offline automatic specialization approaches optimize code prior to the code's deployment through runtime-agnostic optimization techniques. We classify existing offline automatic specialization techniques in four categories. (i) Compile pass selection techniques automatically select the optimal sequence of compiler optimization passes through auto-tuning methods for trying out different combinations [304, 306, 307, 311], (ii) Specialized compiler optimization passes that generate optimized code for specific hardware features such as vectorization [288] or prefetching [375], or for other domain-specific optimizations [7, 239, 267, 345], (iii) Feedback-Driven Optimization (FDO) [73, 104, 175, 175, 179, 203, 302, 308, 342, 362, 395, 442, 460, 463], also known as Profile-Guided Optimization (PGO), optimize programs by repeatedly executing them, collecting runtime

metrics from profiling runs, and using these metrics to guide optimization decisions. However, FDO techniques in their current form typically lack flexibility, as they are optimized primarily for the specific workload on which they were trained, (iv) Specialized Code Tuning approaches either use trial runs to tune specific parameters in the code [238, 317] or apply specialized optimizations to different pieces of the code [317, 402].

While these approaches can significantly improve performance, they fundamentally operate as offline optimization techniques: they generate optimized implementations ahead of deployment based on assumptions about workloads and hardware environments. Consequently, they are often suboptimal when runtime conditions deviate from the scenarios on which they were trained or tuned.

In contrast, Iridescent is designed to dynamically adapt specialization decisions at runtime. Rather than committing to a single statically optimized implementation, Iridescent continuously monitors workload and deployment conditions and selects or generates specialized implementations appropriate for the current execution environment. This enables Iridescent to retain the performance benefits of specialization while avoiding the rigidity of traditional offline approaches, making it more robust to changing workloads and heterogeneous hardware environments.

10.2 Online Automatic Optimizations

We classify existing online automatic optimizations in four categories — (i) runtime code selection, (ii) configuration optimizations, (iii) generic code specializa-

tions, (iv) domain-specific specializations.

Runtime code selection. In this optimization technique, developers pre-generate multiple versions of the code at compile time and then use measured runtime statistics together with offline-trained models to switch between these versions at runtime [92, 214, 391, 394]. While these approaches provide some degree of adaptability, they are fundamentally limited by the need to anticipate and generate all possible code variants *a priori*. As a result, the specialization space is fixed at compile time and cannot incorporate optimization opportunities that emerge only under previously unseen runtime conditions. In contrast, Iridescent can leverage runtime information directly to dynamically generate and apply specializations tailored to the current workload and deployment environment.

Code specializations. Runtime code specialization techniques [27, 86, 87, 103, 161, 321] generate more efficient versions of code by exploiting values and invariants that become known only at runtime. Today, these specializations typically manifest in three forms: (i) Binary rewriting techniques [8]; (ii) Transparent Dynamic Optimization (TDO) systems, such as Dynamo [39], the Transmeta Code Morphing Software [104], and DynamoRIO [59], which optimize code at runtime without requiring application modifications by capturing and optimizing frequently executed instruction traces; and (iii) JIT compiler optimizations for interpreted languages, including type specialization [76, 77, 302] for dynamically typed languages and value specialization [99, 251], in which frequently observed function parameter values are converted into constants. These specializations may additionally be applied selectively depending on the computation

context [130, 202, 276, 316].

In contrast to Iridescent, these approaches are largely generic and are typically applied automatically by the runtime using internal cost models that may not align with application-specific objectives. Moreover, such techniques are generally limited in scope: they optimize localized code regions rather than considering end-to-end system behavior, and they do not leverage domain-specific or deployment-specific specialization opportunities. Iridescent, in contrast, explicitly incorporates workload, deployment, and application context into specialization decisions, enabling coordinated runtime optimizations targeted at improving overall system performance.

Domain-specific specializations. Recently, value specialization techniques have extended beyond interpreted languages to support domain-specific use cases, such as value specialization for GPU kernels [145] and incremental specialization for packet-processing frameworks [127, 272, 343]. While these approaches demonstrate the effectiveness of specialization within specific domains, they are typically tightly coupled to particular systems or optimization scenarios. Iridescent provides a general framework for configuring, applying, exploring, and selecting such specializations at runtime.

Config optimizations. Runtime auto-tuning techniques such as OtterTune [415] perform optimization through measurement-driven trial sessions to identify effective values for different configuration knobs. Some auto-tuning systems, such as OPPerTune [384] and SoftSKU [392], extend this paradigm by performing measurement-driven optimization at runtime without requiring trial phases. How-

ever, these techniques are fundamentally limited to tuning environmental parameters and configuration knobs, as they operate transparently with respect to the deployed application. Consequently, they are not directly applicable to code-level optimizations or specialization. Iridescent extends the measurement-driven auto-tuning paradigm to runtime code specialization, enabling the system to dynamically explore and identify effective combinations of code specializations based on the current workload and deployment environment.

Chapter 11

Combining Offline Prototyping with Online Specialization

Blueprint is designed to construct modular microservice systems from high-level specifications, prioritizing composability, reuse, and systematic exploration of design alternatives. However, this modularity comes with an inherent tradeoff: because users do not directly implement low-level application code, the generated systems are not always optimal for the underlying hardware and workloads. Iridescent provides exactly this code specialization at runtime. In this chapter, we explore how Blueprint and Iridescent can be combined to get the best of both worlds.

11.1 CBDS: Configure, Build, Deploy, Specialize

We combine the CBD methodology from Blueprint with the specialization abilities of Iridescent to introduce the notion of "Configure, Build, Deploy, Specialize" (CBDS) where developers use abstractions of Blueprint to specify the high-level design of the system and generate corresponding implementations which they further specialize by leveraging Iridescent's specifications and mechanisms. Through runtime-aware specialization points, policies, and automated exploration, Iridescent adapts the generated system to observed workloads and environmental conditions. This combination enables a unified development lifecycle in which system structure is defined at design time, while performance and behavior are optimized dynamically at runtime, reducing the need for manual tuning without sacrificing flexibility or efficiency.

As part of this methodology, Iridescent annotations and abstractions are integrated directly into the workflow and wiring specifications of Blueprint. These annotations are used by Blueprint plugins for generating code that can be specialized at runtime. While these specializations are provided as part of the system specification input to Blueprint, they are not directly executed during system generation. Instead, they are only activated once the resulting Blueprint-generated system is deployed and running in its target environment.

This separation enables developers to define the specialization space independently of any concrete implementation, decoupling what can be specialized from how the system is initially constructed. As a result, the same set of special-

izations can be applied transparently across different generated system variants, allowing Iridescent to dynamically optimize multiple implementations without requiring intrusive changes to the application or the plugin implementation.

In a microservice application, developers may wish to specialize low-level code along two orthogonal dimensions: (i) the workflow of the application, and (ii) the scaffolding of the application. Since microservice systems are typically structured around an event-loop model—where an outer loop continuously processes incoming requests and an inner handler executes the logic for each request—both the application workflow and significant portions of the scaffolding are often executed within the request-handling path.

We therefore discuss below the scope of possible specializations in each of these two components, and how they interact within the overall execution structure of the system.

11.1.1 Specializations in Workflow

Specializing workflow code can significantly improve efficiency because workflows often encode high-level control flow that must remain flexible across a variety of deployment scenarios, yet in practice tend to exhibit stable and repetitive patterns at runtime. Generic workflow implementations typically include conditional branches, coordination logic, and fault-handling mechanisms that are not exercised uniformly across executions, leading to unnecessary overhead along the common path.

By specializing workflow code to dominant execution patterns—such as fre-

quently taken branches, common service compositions, or steady-state success cases—the system can streamline control flow, eliminate redundant checks, and reduce coordination overhead. This allows the runtime to optimize for the common-case behavior while still preserving correctness through fallback mechanisms for rare or exceptional execution paths.

11.1.2 Specializations in Scaffolding

Scaffolding generator plugins in Blueprint typically follow a one-size-fits-all policy. Their primary goal is to generate *compatible* and modular code that can support a wide range of workflows, deployment environments, and implementation choices, rather than producing code that is necessarily *optimal* for any single scenario. As a result, plugin-generated scaffolding must preserve abstraction boundaries, configurability, and interoperability across components, which prevents the application of many aggressive compile-time optimizations during code generation.

This emphasis on modularity and generality often introduces additional indirection layers, dynamic dispatch, defensive checks, generic data transformations, and configuration handling that are necessary in the abstract but redundant in a concrete deployment. In practice, however, workloads typically exercise only a narrow subset of this flexibility, repeatedly invoking the same code paths with similar request structures, data shapes, and access patterns. By specializing scaffolding code to these common-case behaviors, the system can remove unnecessary abstraction overheads, streamline data transformations, and tailor resource-

management strategies to observed runtime conditions.

Such specialization enables optimizations that are difficult to safely perform in generic plugin-generated code, including specialization of serialization and deserialization logic, compression and decompression routines, concurrency-control paths, RPC handling for dominant request types, buffer management, and other performance-critical scaffolding operations.

11.2 Raglan: Iridescent as a Blueprint Plugin

This section introduces Raglan, an implementation of Iridescent's abstractions and ideas as a plugin for Blueprint.

11.2.1 Raglan Specialization Runtime

Raglan's specialization runtime performs a function similar to Iridescent's LLVM-based specialization runtime. However, since Go does not have a JIT or extensive support for custom optimization passes, Raglan makes significantly different implementation choices for its specialization runtime. Moreover, the modular nature of Blueprint plugins also adds additional restrictions on the design of Raglan. In this section, I discuss the design and implementation decisions for Raglan's specialization runtime.

11.2.1.1 Emulating a JIT for Go

Since Go doesn't have a JIT, Raglan instead uses Go's `plugin` package for building handlers as shared object files that can be dynamically loaded at runtime.

Go's standard library `plugin` package. Go's `plugin` package allows programs

to load additional functionality at runtime instead by compiling Go code into shared object files (`.so`) and dynamically linking them to a running application. Currently, Go plugins are platform-dependent and supported only on Linux, FreeBSD, and macOS.

Building and Loading Specialized Functions. Like the Iridescent LLVM-based specialization runtime, Raglan also specializes code at the granularity of a single function. Developers place the annotated code to be specialized in a separate file and then dynamically load that file through Go's `plugin` package. Raglan's specialization runtime parses the input function, applies any custom specialization passes, and then builds the specialized code as a standalone shared-object library and dynamically loads the functions from the shared object. Code that intends to use the specialized handler can then obtain a function pointer to the specialized function through the specialization runtime. Like Iridescent's LLVM-based specialization runtime, Raglan also inserts specialization guards and wrapper trampoline functions to ensure that if a specialization check fails, the generic version of the handlers are executed.

Handling Modularity. A Blueprint application may contain multiple specialization points from the workflow specification as well as specialization points in the scaffolding code generated by Blueprint plugins. Consequently, Raglan must support addition of specialization points from both workflow specification code as well as generated code. To do so, Raglan provides a thin interface where each unique location in the code can add a *specialization unit*. Each specialization unit maps to a unique code location in the Blueprint-generated code. A

specialization unit consists of an independent set of functions-to-be specialized and their dependencies in a single Go code source file that can be independently parsed, optimized, and built as a `.so` file. A specialization unit may contain one or more functions-to-be-specialized.

Raglan's specialization runtime collectively manages all these specialization units and generates specialized versions of the functions in each specialization unit. To ensure a holistic view, Raglan currently enforces a restriction of one instance of the specialization runtime per Go process.

11.2.1.2 Custom Specialization Passes

Go Compiler vs LLVM. Compared to LLVM, Go's compiler (cmd/compile) intentionally trades optimization sophistication for simplicity and fast build times, which leads to several key limitations. First, Go's optimization pipeline is smaller and less aggressive, relying heavily on local SSA rewrites rather than deep global analyses, so it lacks advanced techniques like profile-guided optimization, interprocedural optimization across packages, and auto-vectorization. Second, Go performs only limited loop transformations, minimal alias analysis, and relatively conservative inlining decisions, which can leave performance on the table in compute-heavy code. Third, architecture-specific optimizations are narrower in scope, and there's less exploitation of hardware features compared to LLVM's highly tuned backends. For example, Go's compiler does not automatically use vector instructions even when appropriate. Finally, unlike LLVM, Go does not allow reordering of the optimization passes in its optimization pipeline. More-

over, Go also does not support the development or addition of custom compiler passes. Consequently, developers are limited to custom passes that they can implement and run via Go's AST rewriting libraries before Go compilation takes place. Overall, Go prioritizes predictability and compilation speed over extracting every last bit of runtime performance, whereas LLVM is designed to push aggressive, cross-module, and hardware-aware optimizations much further.

Go's support for code rewriting. Go provides strong native support for code rewriting through its standard library, particularly the `go/ast`, `go/parser`, `go/token`, and `go/format packages`. Together, these enable developers to parse source code into Go's Abstract Syntax Tree (AST), inspect and traverse that structure, and make precise programmatic changes to code elements. As the AST preserves the syntactic structure of the original program, transformations are performed safely and deterministically. After transformations, the `go/format` package converts the modified AST back into the source code and ensures that rewritten code adheres to Go's standard formatting conventions, maintaining readability and consistency.

Code rewriting procedures using these packages are typically implemented using APIs provided by Go such as `ast.Inspect` or `astutil.Apply`, where developers can match specific AST node types and rewrite them in place.

Developers implement custom specialization passes using Go's code rewriting packages. These passes operate on a single file unit which contains the annotated handler that the developer wishes to specialize. After the specialization pass is complete, the pass returns a modified version of the file. Raglan supports

the chaining of multiple passes to build custom code rewriting pipelines.

11.2.2 Raglan Implementation

Raglan is implemented in ∼2K Lines of Code consisting mainly of the Specialization Runtime implementation including the Go implementation of Iridescent's APIs, as well as the core JIT-like primitives. In addition, we also modified the behavior of certain existing plugins to add support for Raglan's specialization runtime. Moreover, we also added a new plugin for capturing end-to-end metrics such as throughput and tail latency.

Docker Container Plugin. To integrate Raglan with Blueprint's docker plugin, we had to modify the plugin's `Dockerfile` generation process. Instead of using a distroless base image that simply runs the compiled Go binary application, we had to switch to an image that provides a functional Go runtime so that we can build the `.so` files of specialized functions during execution.

Measurement Plugin. The measurement plugin registers the objective function with Raglan's specialization runtime and provides methods that can be called by the specialization runtime to obtain measurements about the system performance. Currently, the measurement plugin supports service throughput and processing latency as the two possible metrics. When selecting the processing latency, users can configure the plugin to use average latency or n-th percentile latency as the target metric to measure.

Custom Specialization Passes. The current Raglan implementation provides three custom specialization passes: (i) Value specialization pass, which converts

all instances of a variable into a constant (implemented in 70 Lines of Code), (ii) If-Else reorder pass, which reorders the branch conditions of an if-else branch to a specific order (implemented in 135 Lines of Code), and (iii) Struct Field reorder pass, which reorders the fields in a struct definition to reflect a desired layout (implemented in 112 Lines of Code).

11.2.3 Future Extensions

Global System Specializaiton. So far, we have discussed a localized specialization runtime approach where each service process runs its own specialization runtime. A potential extension is to extend the idea of specialization to the full system level instead of keeping specialization decisions localized to single processes. Supporting this would require adopting a sidecar-based design where each service process exposes its local specialization space to a global specialization space responsible for the exploration configuration and specialization decisions based on overall system performance.

Binary Rewriting Passes. A possible future extension is to support optimization passes that operate on the generated binary instead of the Go AST or an intermediate IR. This will circumvent the issue of Go's compiler not allowing for custom optimization passes and allow for optimization passes to modify code close to the final stage of execution. While this approach is plausible, the benefits of such an approach for code optimization are limited.

```go
func GetUserInfo(ctx context.Context, user User) ( Info , error ) {
    if info, err := CheckInNorthAmericaDB(ctx, user); err == nil {
        return info, nil
    } else if info, err := CheckInEuropeDB(ctx, user); err == nil {
        return info, nil
    } else if info, err := CheckInAsiaDB(ctx, user); err == nil {
        return info, nil
    } else if info, err := CheckInSouthAmericaDB(ctx, user); err == nil {
        return info, nil
    } else if info, err := CheckInAfricaDB(ctx, user); err == nil {
        return info, nil
    } else if info, err := CheckInOceaniaDB(ctx, user); err == nil {
        return info, nil
    }
    return nil , NOT_FOUND
}
```

Fig. 11.1: Fixed Order implementation of `GetUserInfo` as generated by Blueprint

11.3 Case Study & Evaluation

In this section, we present a case study of how Blueprint and Iridescent can be integrated through Raglan. We answer the following evaluation questions:

- Can Iridescent dynamically specialize Blueprint-generated implementations to achieve better end-to-end performance than statically generated code?
- Can the integration of Blueprint and Iridescent enable runtime adaptability while preserving the modularity and low developer effort afforded by Blueprint?

11.3.1 Geo-Sharded User Service

For our case study, we consider an application consisting of a `UserService` that serves workloads from users distributed across the world. This case study is inspired by Uber's `UserService` implementation described in "The Tale of Errors in Microservices" [237].

To provide low-latency access and improve scalability, the backing user database is geographically sharded across six continents, with each database instance storing user records only for its corresponding continent. We implement this `UserService` as a Blueprint application. The service exposes two primary APIs: (i) `RegisterUser`, which allows users to create new accounts; and (ii) `GetUserInfo`, which retrieves a user profile from the appropriate continent-specific database using the provided username.

To locate a profile, the `GetUserInfo` API may need to query multiple database instances before finding the correct shard containing the requested user data. Determining the optimal order in which these database instances should be queried is non-trivial, as the best ordering depends not only on the geographic distribution and access patterns of the workload, but also on the runtime latency characteristics of the individual database instances, which may vary over time due to load, contention, or network conditions.

Blueprint Plugin for Optimal Ordering. Since determining an efficient query ordering is a common requirement in distributed applications, we implement a Blueprint plugin called `optorder` that automatically generates different ordering strategies. The plugin exposes multiple policies that developers can select from when generating the application implementation.

Because Blueprint operates purely at generation time and does not have access to runtime workload characteristics or system behavior, the plugin is limited to strategies that either: (i) fix the ordering statically during code generation (as shown in Fig. 11.1); or (ii) generate code capable of dynamically adjusting the or-

dering at runtime using lightweight local heuristics, such as most-recently-used (MRU) access patterns or more optimized scheduling-based approaches such as Smith's Rule ordering. Smith's Rule [376] is a classic scheduling strategy that minimizes weighted completion time by ordering tasks according to the ratio of expected processing cost to expected benefit, making it useful for prioritizing higher-value or more likely operations earlier in execution.

However, because the plugin-generated scaffolding must remain generic and modular, these strategies are fundamentally constrained in how aggressively they can optimize the generated code. Static ordering policies are simple and inexpensive, but they typically perform well only for a narrow range of workloads and can become inefficient when workload characteristics change. Dynamic policies are more adaptable, but selecting an appropriate heuristic, determining how ordering metadata should be maintained, and balancing adaptation overheads against performance gains remain difficult design decisions for developers.

Raglan Integration. To augment this Blueprint plugin with runtime adaptability, we integrate the plugin with Raglan. Instead of relying solely on statically generated orderings or lightweight local heuristics, the plugin exposes the database query ordering itself as a specialization point. Raglan can then dynamically specialize this ordering using observed end-to-end runtime metrics, such as average request latency or tail-latency objectives (e.g., n-th percentile latency), allowing the system to continuously adapt to changing workloads and deployment conditions.

To support runtime reordering, we use the statically fixed order implementation as the baseline implementation and apply Raglan's `ReorderIfPass`. This pass rewrites the branch ordering of the generated fixed order lookup logic according to the ordering selected by Raglan's exploration engine at runtime. In effect, the pass dynamically regenerates the decision structure of the lookup procedure to reflect the currently selected ordering policy.

This integration highlights the complementary strengths of Blueprint and Iridescent. Blueprint provides the modular scaffolding and reusable generation abstractions necessary to construct the distributed application, while Iridescent specializes the generated implementation using runtime feedback to optimize end-to-end behavior. As a result, developers can generate generic, reusable implementations while still benefiting from workload-aware runtime optimization without manually implementing adaptation logic themselves.

11.3.2 Experimental Results

11.3.2.1 Experimental Setup

We deploy the application as an HTTP server and generate open-loop workloads using Poisson request arrivals. T he workload generator is implemented using Blueprint's `workload` plugin.

Each workload is defined as a 6-tuple representing the request distribution across the six continent-specific database branches. For example, a workload configured as (76,10,5,4,3,2) indicates that 76% of requests target North America, 10% target Asia, 5% target Europe, 4% target South America, 3% target Africa, and

2% target Oceania. We consider different workloads for different experiments.

We evaluate static fixed order variants as well as heuristic-based dynamic variants generated by the `optorder` plugin, including a Most-Recently-Used (MRU) policy and a Smith's Rule ordering policy. The Smith's Rule variant updates its ordering every 1 minute based on observed execution characteristics over a 15-second monitoring period.

Finally, we evaluate a Raglan-enabled variant that performs runtime exploration of ordering configurations using end-to-end performance metrics. We configure the exploration phase duration to 15 seconds, with a 1-second monitoring window for each evaluated configuration. The exploration is triggered periodically every minute. We intentionally choose this relatively aggressive exploration frequency to stress-test the adaptability of the system. In realistic deployments, however, exploration would typically occur at much coarser timescales and would likely be triggered by developer-defined conditions or workload-shift detection mechanisms rather than fixed periodic intervals alone. This would reduce exploration overhead while still allowing the system to adapt to meaningful changes in workload behavior or deployment conditions.

11.3.2.2 Plugin Complexity

Since all policies share a common plugin implementation for integrating with Blueprint, the primary differences between policy implementations lie in the code templates used to instantiate each ordering strategy. The fixed-order policy has the smallest template, consisting of 38 lines of code, and serves as the base for all other policies.

The MRU policy extends this template with 20 additional lines of code, primarily to support correct handling of concurrent updates to the ordering under ongoing request traffic. The Smith's Rule policy adds 44 lines of code, including a 39-line function that continuously updates the ordering based on runtime statistics and a 5-line instrumentation that measures per-database call frequency and latency to drive the ordering decisions.

In contrast, Raglan's ordering policy extends the fixed-order template with 27 additional lines of code. Of these, 26 lines are used to interface with the specialization runtime and to define and register the ordering as a specialization point, while the remaining line initializes the statically generated exploration space of possible orderings since that information is available at code generation time.

Using this plugin together with Raglan's runtime, developers can transparently optimize for arbitrary combinations of end-to-end metrics without requiring any further modifications to the plugin implementation itself.

11.3.2.3 Quantitative Results

Optimizing Average Latency. We consider three workload configurations: W1=(75,5,5,5,5,5), W2=(5,5,5,5,5,75), and W3=(50,45,2,1,1,1). We execute W1, W2, and W3 in sequence with a request rate of 1000 requests/s, with each workload lasting 1 minute. We first execute workload W1, and at the 1-minute mark switch to W2, followed by W3 at the 2-minute mark.

We repeat this execution pattern for all evaluated variants: fixed-order baselines, MRU dynamic baseline, Smith's Rule dynamic baseline, and the Raglan-

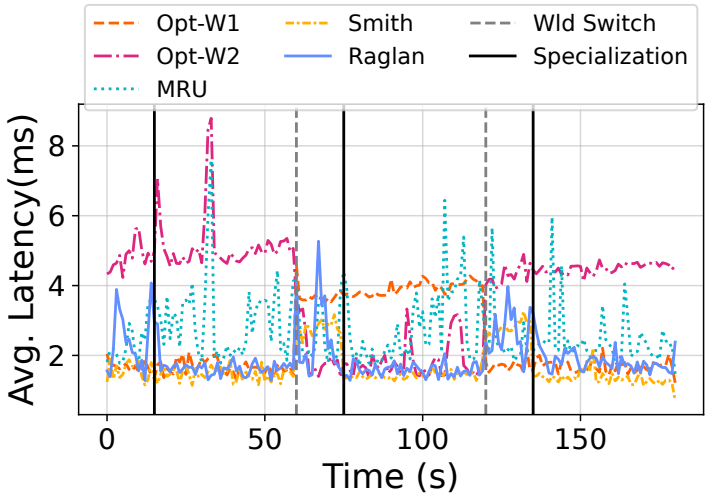


Fig. 11.2: Online Adaptation for the UserService case study with Raglan in comparison to other baseline configurations

enabled variant. For the fixed order baselines, we use the `optorder` plugin to generate fixed-order variants manually selected for W1 and W2. We refer to these implementations as Opt-W1 and Opt-W2, corresponding to the statically optimal orderings for workloads W1 and W2, respectively. We note that the optimal static ordering for workload W3 coincides with that of W1 for minimal variation in response latencies across database instances. For the Raglan-enabled variant, we configure Raglan to optimize for end-to-end average latency using Raglan's measurement plugin. We measure end-to-end request latency and report the average latency over 1-second sliding windows.

Fig. 11.2 shows that Opt-W1 and Opt-W2 achieve the best performance for the specific workloads they are manually optimized for, but suffer significant degradation—up to a 3.2× increase in average end-to-end latency—when eval-

uated outside their target workload regime. MRU performs poorly because it relies only on recently observed accesses, causing it to overreact to short-term fluctuations and oscillate between suboptimal orderings rather than converging to a stable, globally efficient configuration.

Among the baselines, Smith's Rule ordering performs best, as expected, because it explicitly prioritizes database probes based on their estimated cost-to-benefit ratio, thereby minimizing expected average lookup latency across heterogeneous database response times. However, this performance comes at the cost of increased implementation complexity, since Smith's Rule requires continuous tracking and maintenance of per-database latency statistics and their ongoing recomputation under changing workload conditions.

In contrast, the Raglan-based variant consistently maintains low average latency across all workloads while achieving near-optimal performance with reduced implementation effort. Moreover, it adapts automatically to workload transitions without requiring explicit per-database latency tracking or manual policy updates, demonstrating the benefit of runtime specialization over static or manually engineered ordering strategies. Consequently, it achieves similar performance to the theoretical best approach of Smith's Rule.

Beyond Optimizing Average Latency. We next consider the setting where developers instead optimize for the 90th percentile latency. In this experiment, we additionally introduce heterogeneous database response latencies to better reflect realistic deployment conditions where different regions may exhibit varying network and processing delays. We execute a single workload, W4=(85,6,9,0,0,0),

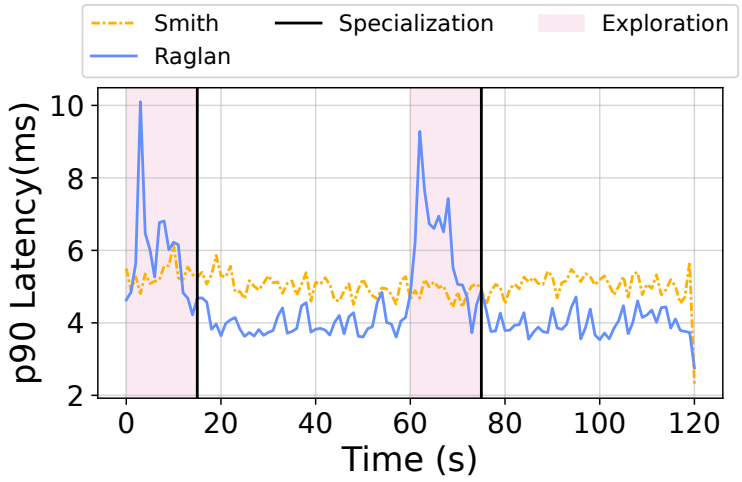


Fig. 11.3: Raglan exploration comparison with Smith's Rule with minimizing 90th percentile latency as the objective function

for a duration of two minutes for both the Raglan-enabled variant and the Smith's Rule ordering variant.

Fig. 11.3 shows that when Raglan is configured to directly optimize a percentile-based objective (90th percentile latency) during exploration, it can outperform Smith's Rule. This behavior arises because Smith's Rule is fundamentally designed to optimize a global average-cost objective derived from fixed task weights, making it effective for minimizing average-case latency but less responsive to rare, high-cost execution paths that dominate tail latency. In contrast, Raglan's exploration engine can directly optimize percentile-based metrics as the objective function, explicitly rewarding specialization configurations that reduce worst-case or near-worst-case request latencies.

As a result, Raglan can preferentially select specialization configurations that eliminate or shorten tail-heavy execution paths, even when such choices do not

improve—and may occasionally degrade—average-case performance. By directly optimizing distributional objectives rather than relying on fixed local heuristics, Raglan can therefore discover specialization strategies that are better aligned with tail-latency goals than Smith's Rule.

Moreover, changing the optimization target from average latency to an n-th percentile latency metric requires only a one-line modification to the wiring specification of the application and does not require any changes to the `optorder` plugin implementation itself.

11.3.2.4 Results Summary

We have demonstrated that integrating Iridescent with Blueprint enables dynamic specialization of generated systems that improves performance over statically generated implementations. The combined system achieves performance comparable to optimal dynamic baselines when such baselines exist, and can further exceed them in settings where no known optimal strategy is available. Importantly, these gains are achieved without requiring additional implementation effort or manual modification from users, highlighting how runtime specialization can complement generation-time system construction to deliver both flexibility and performance.

11.4 Summary

In this chapter, we have shown that Blueprint and Iridescent operate at different and complementary levels of abstraction. Due to this, the two approaches can effectively combine together and can yield significant performance boosts.

In this chapter, we have demonstrated through a case study the potential benefits of combining these approaches. We believe that this combination gives rise to a new design, implementation, optimization pipeline where users can automatically generate runnable systems from heterogeneous components and have individual components be specialized for the workload. Moreover, this combination also opens up opportunities for end-to-end automatic system optimization and specialization instead of specializations restricted to individual components.

Part IV

Conclusion

Chapter 12

Conclusion

12.1 Summary of Results

In this dissertation, I proposed and evaluated Human-in-the-Loop Specification-Driven Automation (HITLSDA) techniques for reducing the manual effort required to perform cloud system lifecycle tasks. The central insight underlying HITLSDA is the separation of the *creative* aspects of these tasks from their corresponding *mechanical* execution. Developers are typically well suited to specifying high-level goals, constraints, and design intent, while much of the repetitive exploration, transformation, optimization, and execution process can be automated. The techniques presented in this dissertation leverage this observation by introducing abstractions that allow developers to programmatically express the creative component of lifecycle tasks, together with automation frameworks that automatically carry out the associated mechanical processes.

This dissertation makes the following concrete contributions:

Rapid Offline Prototyping with Blueprint. With Blueprint (Chapter 4), I designed a toolchain for generating easily reconfigurable microservice systems. Blueprint reduces manual effort for exploring the design space of microservice systems for use-cases that involve configuring, building, and deploying variants of microservice applications with modified designs. Blueprint instantiates as a HITLSDA technique to provide a set of abstractions through which users can programmatically specify the creative component, *i.e.* the design of the system they want to try, which Blueprint's compiler uses to automate the mechanical component of converting the design into a complete, deployable implementation of the system. Blueprint substantially eases development and reconfiguration by reducing the lines of code that a developer needs to manually write for carrying out design selection and implementation tasks.

Online Specialization of Systems with Iridescent. With Iridescent (Chapter 8), I designed a framework for enabling online workload-driven runtime code specialization of systems to optimize and improve end-to-end system performance. Iridescent provides a toolkit for developers to implement system-specific specializations that utilize runtime data and invariants to generate more performant systems. Iridescent instantiates as a HITLSDA technique to provide the necessary abstractions and tooling for developers to programmatically specify the creative component, *i.e.* the specialization space of the system, which Iridescent's runtime uses to automate the mechanical component of exploring the space of potential specializations to find the best performing configuration. By

combining developer insight with automated compiler optimizations, Iridescent enables systems to adapt efficiently to changing workloads and environments with minimal manual effort.

12.2 Future Work

I now discuss several directions in which this work could be extended in the future. In particular, I believe there are three immediately promising directions: (i) developing HITLSDA techniques for problematic behavior diagnosis and system evolution; (ii) generating representative large-scale microservice testbeds; and (iii) combining HITLSDA techniques with agentic AI systems to further automate cloud lifecycle tasks.

12.2.1 HITLSDA for Problem Diagnosis and System Evolution

This dissertation focuses on two specific tasks of the cloud lifecycle – (i) Selecting the best design; and (ii) Optimizing for best performance. The cloud lifecycle has two more tasks (§2.2) that elicit significant manual effort – (i) Understanding problematic behaviors; and (ii) Evolving and improving systems.

Developing automation techniques for understanding problematic behaviors is challenging for several reasons. First, the space of possible problematic behaviors is both large and highly diverse. Failures and anomalies can manifest in many different forms, often requiring fundamentally different analysis and remediation strategies, making it difficult to develop a single unified approach.

Second, cloud systems generate massive volumes of observability data, forcing operators to sift through large amounts of logs, traces, metrics, and events to first identify the relevant information and then determine the underlying root cause. Third, the necessary observability data may itself be incomplete or unavailable because detailed instrumentation is often omitted due to performance overheads or because developers did not anticipate the relevant failure mode. Finally, the scale and complexity of modern cloud systems dramatically expand the space of potential root-cause locations, since problematic behavior may emerge from subtle interactions across multiple components, layers, and services rather than from a single isolated fault.

Developing automation techniques for evolving and improving systems is difficult for multiple reasons. First, identifying the locations within the system where modifications must be applied is itself non-trivial. Required changes may span multiple abstraction levels, including implementation details, service interfaces, deployment configurations, or even the overall system architecture. Second, developers must manually design and apply these modifications, and incorrectly applying changes can introduce regressions, degrade performance, or break existing functionality. Finally, evolving a system also requires updating or generating new tests to validate both the correctness and performance of the modified system, further increasing the manual effort required for system evolution.

HITLSDA techniques have the potential to substantially reduce the manual effort required to understand problematic behaviors and evolve cloud sys-

tems. By separating high-level intent from low-level execution, such techniques could help developers automatically extract relevant insights from massive volumes of observability data, localize likely root causes across complex multi-component systems, and dynamically identify missing instrumentation necessary for diagnosing failures. Rather than requiring operators to manually correlate logs, traces, and metrics across layers of the system, HITLSDA-based approaches could automate much of this mechanical analysis while allowing developers to guide the process using domain-specific specifications and debugging objectives.

Similarly, for system evolution, HITLSDA techniques could assist developers in identifying the precise locations where changes must be applied, whether at the code, interface, deployment, or architectural level. Such systems could further help developers automatically generate candidate patches, prototype implementation alternatives, and validate modifications against existing functionality and performance requirements. By automating repetitive transformation and testing tasks while keeping developers in control of the high-level design intent, these techniques could significantly reduce the complexity and risk associated with evolving large cloud systems. Developing such HITLSDA techniques represents an important direction for future research.

12.2.2 Representative Large-Scale Microservice Testbeds

Validating and testing new advancements for microservices often requires developers to experimentally modify systems by introducing new design decisions,

architectural choices, algorithms, backend components, or execution strategies in order to evaluate their impact on system behavior. Ideally, developers and researchers would conduct such experiments directly on production microservice deployments to make realistic claims about scalability, robustness, and generalizability. However, access to production systems is typically restricted to a small number of organizations. Even when such access is available, experimentation must be carefully constrained to avoid disrupting production workloads, limiting both the scope and aggressiveness of possible evaluations.

Consequently, researchers often rely on open-source microservice benchmarks as experimental testbeds, as discussed in §3.2. While these systems provide representative examples of common microservice design patterns and implementation strategies, they do not adequately capture the scale, heterogeneity, complexity, and operational characteristics of large production-grade cloud services [366]. As a result, experiments and analyses conducted solely on these open-source systems are often insufficient for making broad claims about the effectiveness, scalability, or general applicability of proposed techniques.

While Blueprint reduces the effort required to explore the design space of microservice systems, on its own, Blueprint is not enough to build large-scale microservice benchmarks as a researcher must still manually implement the large number of services and correctly connect them. Even if a team of researchers manually did that, the produced system will not necessarily be representative of the production systems. As a result, constructing flexible representative large-scale microservice benchmarks remains an unsolved problem.

While Blueprint does not directly solve this problem, Blueprint's abstractions are the necessary building block for solving this problem. To solve this problem, we have proposed Palette [21], a HITLSDA tool that combines open-source tracing datasets with a novel experimentation library built on top of Blueprint's abstractions that allow users to programmatically specify the creation of large-scale microservice benchmarks from representative tracing datasets provided by large cloud companies. With the novel experimentation library, the users can specify which structural and performance features of the system observable from the tracing dataset they would like to preserve, and the underlying generator built on Blueprint's compiler automatically generates a microservice system that preserves the representative features.

12.2.3 Enabling Further Automation with Agentic AI

End-to-end automation of cloud systems has been a long standing goal for developers and operators. However, such automation of cloud tasks has been elusive for multiple reasons. First, designing and maintaining systems is a long and arduous process which requires correctly handling a myriad of complex and intertwined requirements [151]. Second, cloud systems tend to be large and complex which prevents easy operation and understanding as it is impossible for any single operator to have full insight into the operational context of the system [141, 150]. Finally, the development of cloud standards and mature automation tooling for various cloud tasks has required significant effort from both industry practitioners and researchers and still continues to be an open area of

investigation and improvement.

In this dissertation, we have shown how we can partially automate large parts of the cloud lifecycle. We believe that we are now on the cusp of achieving the elusive goal of end-to-end automation. This is for two reasons. First, there has been a confluence of standardization and automation tools, such as Kubernetes for deployment, maintenance, and convergence of systems to desired states [367]; OpenTelemetry for monitoring and observability via logs, traces, and metrics; and automatic bug-finding and verification tools that can be utilized in both development and production [53, 58, 105]. Second, the recent rise of Large Language Models (LLMs) and Agentic AI has provided developers and operators with increasingly sophisticated automatic coding and code understanding tools through compound AI systems [449] and autonomous agentic systems [26].

Intent-based System Design and Operation. We have introduced Intent-based System Design and Operation [18], a vision for automating cloud lifecycle tasks by combining the generative capabilities of LLMs and agentic AI systems with the HITLSDA tools introduced in this dissertation. Inspired by intent-based networking [82], where networks automatically configure themselves to satisfy operator intent, this approach allows developers to specify the desired *functional* and *operational* requirements of a system directly in natural language. HITLSDA tools augmented with LLMs and agentic AI can then automatically infer, synthesize, and execute the corresponding implementation and operational workflows required to realize the specified intent.

For example, we are building Cerulean [16], a HITLSDA system that com-

bines LLMs with the abstractions and compiler infrastructure of Blueprint to automate the generation of microservice systems from high-level developer specifications. By leveraging LLM reasoning together with structured system abstractions, Cerulean reduces the manual effort involved in designing service architectures, generating boilerplate components, configuring scaffolding, and exploring implementation alternatives.

Combining the generative powers of LLMs and Agentic AI with the principled approach of HITLSDA has the potential of opening up new lines of research into automating cloud lifecycle tasks.

Bibliography

[1] Dapr: Distributed application runtime. https://dapr.io/.

[2] A. S. Abdelfattah. Student research abstract: Microservices-based systems visualization. *arXiv preprint arXiv:2207.12998*, 2022.

[3] AcmeAir. Acmeair. https://github.com/acmeair.

[4] S. Agarwal, R. Sinha, G. Sridhara, P. Das, U. Desai, S. Tamilselvam, A. Singhee, and H. Nakamuro. Monolith to microservice candidates using business functionality inference. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 758–763. IEEE, 2021.

[5] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[6] I. K. Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan. A model-driven architecture for automated deployment of microservices. *Applied Sciences*, 11(20):9617, 2021.

[7] C. L. Alappat, J. Seiferth, G. Hager, M. Korch, T. Rauber, and G. Wellein. Yasksite: stencil optimization techniques applied to explicit ode methods on modern architectures. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 174–186. IEEE, 2021.

[8] A. A. Ali and E. Rohou. Dynamic function specialization. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 163–170. IEEE, 2017.

[9] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[10] S. Allen, M. Toslali, S. Parthasarathy, F. Oliveira, and A. K. Coskun. Tritium: A cross-layer analytics system for enhancing microservice rollouts in the cloud. In *Proceedings of the Seventh International Workshop on Container Technologies and Container Clouds*, pages 19–24, 2021.

[11] N. Alshuqayran. *Static Microservice Architecture Recovery Using Model-Driven Engineering*. PhD thesis, University of Brighton, 2020.

[12] M. Amaral, T. Chiba, S. Trent, T. Yoshimura, and S. Choochotkaew. Microlens: A performance analysis framework for microservices using hidden metrics with bpf. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 230–240. IEEE, 2022.

[13] L. An, A.-J. Tu, X. Liu, and R. Akkiraju. Real-time statistical log anomaly detection with continuous aiops learning. In *CLOSER*, pages 223–230, 2022.

[14] V. Anand, D. Garg, and A. Kaufmann. Iridescent: A framework enabling online system implementation specialization. *arXiv preprint arXiv:2508.16690*, 2025.

[15] V. Anand, D. Garg, A. Kaufmann, and J. Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 482–497, 2023.

[16] V. Anand, A. Kumbhare, C. Irvene, C. Bansal, G. Somashekar, J. Mace, P. Las-Casas, R. Bianchini, and R. Fonseca. Automated service design with cerulean (project showcase). *6th International Workshop on Cloud Intelligence/AIOps Workshop (co-located with ICSE)*, 2025.

[17] V. Anand, P. Las-Casas, R. Fonseca, and A. Kaufmann. Towards using llms for distributed trace comparison (abstract). *6th International Workshop on Cloud Intelligence/AIOps Workshop (co-located with ICSE)*, 2025.

[18] V. Anand, Y. Li, A. G. Kumbhare, C. Irvene, C. Bansal, G. Somashekar, J. Mace, P. Las-Casas, R. Bianchini, and R. Fonseca. Intent-based system design and operation. *Practical Adoption Challenges of ML for Systems (PACMI'25) (co-located with SOSP)*, 2025.

[19] V. Anand and J. Mace. Deathstarbench - modified with x-trace. https://github.com/JonathanMace/DeathStarBench/.

[20] V. Anand, M. Stolet, T. Davidson, I. Beschastnikh, T. Munzner, and J. Mace. Aggregate-driven trace visualizations for performance debugging. *arXiv preprint arXiv:2010.13681*, 2020.

[21] V. Anand, M. Stolet, J. Mace, and A. Kaufmann. Generating representative macrobenchmark microservice systems from distributed traces with palette. *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2025)*, 2025.

[22] V. Anand and J. Wonsil. Tracey-distributed trace comparison and aggregation using nlp techniques.

[23] V. Anand, Z. Xie, M. Stolet, R. De Viti, T. Davidson, R. Karimipour, and J. Mace. The odd one out: Energy is not like other metrics. *HotCarbon*, 2022.

[24] M. V. d. A. ANDRADE. Migração de microsserviços containerizados em tempo de execução. Master's thesis, Universidade Federal de Pernambuco, 2018.

[25] T. Angerstein, C. Heger, A. v. Hoorn, D. Okanović, H. Schulz, S. Siegl, and A. Wert. Continuity-automatisiertes performance-testen in der kontinuierlichen softwareentwicklung: Abschlussbericht. 2020.

[26] Anthropic Blog. Building effective agents. Accessed July 2025 from https://www.anthropic.com/engineering/building-effective-agents, 2024.

[27] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, 2000.

[28] V. Arya, K. Shanmugam, P. Aggarwal, Q. Wang, P. Mohapatra, and S. Nagar. Evaluation of causal inference techniques for aiops. In *8th ACM IKDD CODS and 26th COMAD*, pages 188–192. 2021.

[29] asc lab. Asclab micronaut poc - lab insurance sales portal. https://github.com/asc-lab/micronaut-microservices-poc/.

[30] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 229–236, 2021.

[31] W. K. Assunção, J. Krüger, and W. D. Mendonça. Variability management meets microservices: six challenges of re-engineering microservice-based webshops. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–6, 2020.

[32] S. Athlur, N. Sondhi, S. Batra, S. Kalambur, and D. Sitaram. Cache characterization of workloads in a microservice environment. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 45–50. IEEE, 2019.

[33] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In *European Conference on Software Architecture*, pages 159–174. Springer, 2018.

[34] A. Aznavouridis, K. Tsakos, and E. G. Petrakis. Micro-service placement policies for cost optimization in kubernetes. In *International Conference on Advanced Information Networking and Applications*, pages 409–420. Springer, 2022.

[35] Azure. Noisy neighbour. Retrieved Septemeber 2022 from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor, 2022.

[36] Azure. Technology choices for azure solutions. Retrieved September 2022 from https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/technology-choices-overview, 2022.

[37] A. F. Baarzi and G. Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441, 2021.

[38] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[39] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000.

[40] A. Baluta, J. Mukherjee, and M. Litoiu. Machine learning based interference modelling in cloud-native applications. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 125–132, 2022.

[41] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.

[42] G. C. Baptista et al. *Failure injection in microservice applications*. PhD thesis, Universidade de Coimbra, 2021.

[43] Barber0. Deathstarbench social network grpc. Retrieved August 2022 from https://github.com/Barber0/DeathStarBench/tree/sn_grpc.

[44] Barber0. Deathstarbench no tracing. Retrieved August 2022 from https://github.com/Barber0/DeathStarBench/tree/no_tracing, 2021.

[45] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[46] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano. Neptune: Network-and gpu-aware management of serverless functions at the edge. *arXiv preprint arXiv:2205.04320*, 2022.

[47] J. Berg, F. Ruffy, K. Nguyen, N. Yang, T. Kim, A. Sivaraman, R. Netravali, and S. Narayana. Snicket: Query-driven distributed tracing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 206–212, 2021.

[48] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020.

[49] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[50] S. Bhatia, C. Consel, A.-F. Le Meur, and C. Pu. Automatic specialization of protocol stacks in operating system kernels. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 152–159. IEEE, 2004.

[51] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien. Migration from monolith to microservices: Benchmarking a case study. 2020.

[52] M. Bogo, J. Soldani, D. Neri, and A. Brogi. Fine-grained management of cloud-native applications, based on tosca. *Internet Technology Letters*, 3(5):e212, 2020.

[53] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. 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*, pages 836–850, 2021.

[54] A. Brogi, A. Corradini, and J. Soldani. Estimating costs of multi-component enterprise applications. *Formal Aspects of Computing*, 31(4):421–451, 2019.

[55] R. Brondolin and M. D. Santambrogio. A black-box monitoring approach to measure microservices runtime performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.

[56] R. Brondolin and M. D. Santambrogio. Presto: a latency-aware power-capping orchestrator for cloud-native microservices. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 11–20. IEEE, 2020.

[57] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 221–227, 2021.

[58] M. Brooker and A. Desai. Systems correctness practices at aws: Leveraging formal and semi-formal methods. *Queue*, 22(6):79–96, 2024.

[59] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[60] V. Bushong, D. Das, A. Al Maruf, and T. Cerny. Using static analysis to address microservice architecture reconstruction. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1201. IEEE, 2021.

[61] V. Bushong, D. Das, and T. Cernỳ. Reconstructing the holistic architecture of microservice systems using static analysis. In *CLOSER*, pages 149–157, 2022.

[62] A. Byrne, S. Nadgowda, and A. K. Coskun. Ace: Just-in-time serverless software component discovery through approximate concrete execution. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 37–42, 2020.

[63] S. Caculo, K. Lahiri, and S. Kalambur. Characterizing the scale-up performance of microservices using teastore. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 48–59. IEEE, 2020.

[64] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 29–39, 2022.

[65] M. Camilli, A. Janes, and B. Russo. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software*, 187:111225, 2022.

[66] M. Camilli and B. Russo. Modeling performance of microservices systems with growth theory. *Empirical Software Engineering*, 27(2):1–44, 2022.

[67] L. Cao and P. Sharma. Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 168–174, 2021.

[68] W. Cao, Z. Cao, and X. Zhang. Research on microservice anomaly detection technology based on conditional random field. In *Journal of Physics: Conference Series*, volume 1213, page 042016. IOP Publishing, 2019.

[69] V. Cardellini, E. Casalicchio, S. Iannucci, M. Lucantonio, S. Mittal, D. Panigrahi, and A. Silvi. An intrusion response system utilizing deep q-networks and system partitions. *arXiv preprint arXiv:2202.08182*, 2022.

[70] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *PPSC*, pages 400–405. Citeseer, 1989.

[71] A. Carrusca, M. C. Gomes, and J. Leitão. Microservices management on cloud/edge environments. In *International Conference on Service-Oriented Computing*, pages 95–108. Springer, 2019.

[72] C. Cassé, P. Berthou, P. Owezarski, and S. Josset. Using distributed tracing to identify inefficient resources composition in cloud applications. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 40–47. IEEE, 2021.

[73] D. Chen, D. X. Li, and T. Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.

[74] N. Chen, H. Tu, X. Duan, L. Hu, and C. Guo. Semisupervised anomaly detection of multivariate time series based on a variational autoencoder. *Applied Intelligence*, pages 1–25, 2022.

[75] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran. {CLARION}: Sound and clear provenance tracking for microservice deployments. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3989–4006, 2021.

[76] M. Chevalier-Boisvert and M. Feeley. Interprocedural type specialization of javascript programs without type analysis. *arXiv preprint arXiv:1511.02956*, 2015.

[77] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing matlab through just-in-time specialization. In *International Conference on Compiler Construction*, pages 46–65. Springer, 2010.

[78] B. Choi, J. Park, C. Lee, and D. Han. phpa: A proactive autoscaling framework for microservice chain. In *5th Asia-Pacific Workshop on Networking (APNet 2021)*, pages 65–71, 2021.

[79] N. Chondamrongkul, J. Sun, and I. Warren. Formal security analysis for software architecture design: An expressive framework to emerging architectural styles. *Science of Computer Programming*, 206:102631, 2021.

[80] S. Choochotkaew, T. Chiba, S. Trent, and M. Amaral. Run wild: Resource management system with generalized modeling for microservices on cloud. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 609–618. IEEE, 2021.

[81] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu. Deeprest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 181–198, 2022.

[82] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. Rfc 9315: Intent-based networking - concepts and definitions, 2022.

[83] Y. Coady, R. Cox, J. DeTreville, P. Druschel, J. Hellerstein, A. Hume, K. Keeton, T. Nguyen, C. Small, L. Stein, et al. Green team paper: Falling off the cliff: when systems go nonlinear. In *10th Workshop on Hot Topics in Operating Systems, HotOS 2005*, 2005.

[84] A. Cockcroft. The evolution of microservices. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[85] A. Cockcroft. Microservices workshop: Why, what, and how to get there. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, 2016.

[86] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation: International Seminar Dagstuhl Castle, Germany, February 12–16, 1996 Selected Papers*, pages 54–72. Springer, 2005.

[87] C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, 1996.

[88] A. Cornacchia, V. Anand, M. Bilal, Z. A. Qazi, and M. Canini. Dmas-forge: A framework for transparent deployment of ai applications as distributed systems. *1st Workshop on Systems for Agentic AI (SAA'25) (co-located with SOSP)*, 2025.

[89] S. P. E. Corporation. Spec accel benchmark suite. Available from https://www.spec.org/accel, 2023.

[90] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *Journal of Systems and Software*, 183:111084, 2022.

[91] V. Cortellessa and L. Traini. Detecting latency degradation patterns in service-based systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 161–172, 2020.

[92] D. Costa and A. Andrzejak. Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26, 2018.

[93] C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson. Automated performance prediction of microservice applications using simulation. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.

[94] C. Cui, G. Wu, W. Chen, J. Zhu, and J. Wei. Feedback-based, automated failure testing of microservice-based applications. *arXiv preprint arXiv:1908.06466*, 2019.

[95] J. Curtis. On language-agnostic abstract-syntax trees: student research abstract. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1619–1625, 2022.

[96] Q. N. Dao and T. Hey. *Coreference Resolution for Software Architecture Documentation*. PhD thesis, Informatics Institute, 2022.

[97] N. Daw, U. Bellur, and P. Kulkarni. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 585–599, 2021.

[98] E. De Angelis, A. Pellegrini, and M. Proietti. Automatic extraction of behavioral features for test program similarity analysis. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 129–136. IEEE, 2021.

[99] I. R. de Assis Costa, H. N. Santos, P. R. Alves, and F. M. Q. Pereira. Just-in-time value specialization. *Computer Languages, Systems & Structures*, 40(2):37–52, 2014.

[100] A. De Iasio and E. Zimeo. A framework for microservices synchronization. *Software: Practice and Experience*, 51(1):25–45, 2021.

[101] R. De Viti, V. Anand, P. Ingo, and D. Garg. Sok: Demystifying the multiverse of mpc protocols. *arXiv preprint arxiv:2512.11699*, 2025.

[102] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[103] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, 1995.

[104] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing/spl trade/software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 15–24. IEEE, 2003.

[105] P. Deligiannis, N. Ganapathy, A. Lal, and S. Qadeer. Building reliable cloud services using p#(experience report). *arXiv preprint arXiv:2002.04903*, 2020.

[106] delimitriou. Deathstarbench social network application refactor for better performance. Retrieved August 2022 from https://github.com/delimitrou/DeathStarBench/pull/101, 2021.

[107] B. Deng, W. Wu, and L. Song. Redundant logic elimination in network functions. In *Proceedings of the Symposium on SDN Research*, pages 34–40, 2020.

[108] U. Desai, S. Bandyopadhyay, and S. Tamilselvam. Graph neural network to dilute outliers for refactoring monolith application. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 72–80, 2021.

[109] D. Di Pompeo, M. Tucci, A. Celi, and R. Eramo. A microservice reference case study for design-runtime interaction in mde. In *STAF (Co-Located Events)*, pages 23–32, 2019.

[110] M. Dinga. An empirical evaluation of the energy and performance overhead of monitoring tools on docker-based systems. Retrived August 2022 from https://github.com/MadalinaDinga/thesis-2022-monitoring-tools-integration-with-train-ticket-replication-package/, 2022.

[111] P. Dogga, K. Narasimhan, A. Sivaraman, S. Saini, G. Varghese, and R. Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. *Proceedings of Machine Learning and Systems*, 4:601–622, 2022.

[112] dot-net architecture. eshoponcontainers. https://github.com/dotnet-architecture/eShopOnContainers.

[113] DPDK Project. Data plane development kit. http://www.dpdk.org/, 2022. Retrieved Feb 2, 2022.

[114] T. F. Düllmann and A. Van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th acm/spec on international conference on performance engineering companion*, pages 171–172, 2017.

[115] T. F. Düllmann, A. van Hoorn, V. Yussupov, P. Jakovits, and M. Adhikari. Ctt: Load test automation for tosca-based cloud applications. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pages 89–96, 2022.

[116] EdwinVW. Pitstop: Garage management system. https://github.com/EdwinVW/pitstop/.

[117] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn. Microservices: A performance tester's dream or nightmare? In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 138–149, 2020.

[118] A. El Malki and U. Zdun. Evaluation of api request bundling and its impact on performance of microservice architectures. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 419–424. IEEE, 2021.

[119] S. Elflein, I. S. Kounev, J. Grohmann, and S. Eismann. Automatic identification of parametric dependencies for performance models.

[120] M. Elsaadawy, M. Younis, J. Wang, X. Hou, and B. Kemme. Dynamic application call graph formation and service identification in cloud data centers. *IEEE Transactions on Network and Service Management*, 2022.

[121] D. Elsner, D. Bertagnolli, A. Pretschner, and R. Klaus. Challenges in regression test selection for end-to-end testing of microservice-based software systems. In *International Conference on Automation of Software Test (AST'22)*, 2022.

[122] A. Erradi, W. Iqbal, A. Mahmood, and A. Bouguettaya. Web application resource requirements estimation based on the workload latent features. *IEEE Transactions on Services Computing*, 2019.

[123] A. M. Espinoza, R. Wood, S. Forrest, and M. Tiwari. Back to the future: N-versioning of microservices. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–427. IEEE, 2022.

[124] A. Farley. Continuous performance testing of faas and microservices based on tosca topology and orchestration specifications. B.S. thesis, 2020.

[125] J. Faro. Deathstarbench metastability failure fork. Retrieved August 2022 from https://github.com/jfaro/DeathStarBench, 2022.

[126] J. Faro. Metastable failure vulnerability testing for deathstarbench's social network microservice benchmark. Retrieved August 2022 from https://github.com/jfaro/metastable-failures, 2022.

[127] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. Packetmill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–17, 2021.

[128] K. Fedyuk. *Sheik: Dynamic location and binding of microservices for cloud/edge settings*. PhD thesis, NOVA University of Lisbon, 2020.

[129] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes. My services got old! can kubernetes handle the aging of microservices? In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 40–47. IEEE, 2021.

[130] O. Flückiger, G. Chari, M.-H. Yee, J. Ječmen, J. Hain, and J. Vitek. Contextual dispatch for function specialization. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–24, 2020.

[131] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[132] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.

[133] J. Forough, M. Bhuyan, and E. Elmroth. Detection of vsi-ddos attacks on the edge: A sequential modeling approach. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.

[134] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840, 2021.

[135] A. Fuerst, A. Ali-Eldin, P. Shenoy, and P. Sharma. Cloud-scale vm-deflation for running interactive applications on transient servers. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 53–64, 2020.

[136] N. Fukuda, C. Wu, S. Horiuchi, and K. Tayama. Fault report generation for ict systems by jointly learning time-series and text data. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2022.

[137] S. Galantino, M. Iorio, F. Risso, and A. Manzalini. Raygo: Reserve as you go. In *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 269–275. IEEE, 2021.

[138] Y. Gan, M. Liang, D. L. Sundar Dev, and C. Delimitrou. Sage: Practical & scalable ml-driven performance debugging in microservices.

[139] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. 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*, pages 3–18, 2019.

[140] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[141] V. Ganatra, A. Parayil, S. Ghosh, Y. Kang, M. Ma, C. Bansal, S. Nath, and J. Mace. Detection is better than cure: A cloud incidents perspective. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1891–1902, 2023.

[142] D. Garg, N. C. Narendra, and S. Tesfatsion. Heuristic and reinforcement learning algorithms for dynamic service placement on mobile edge cloud. *arXiv preprint arXiv:2111.00240*, 2021.

[143] P. Genfer and U. Zdun. Identifying domain-based cyclic dependencies in microservice apis using source code detectors. In *European Conference on Software Architecture*, pages 207–222. Springer, 2021.

[144] P. Genfer and U. Zdun. Avoiding excessive data exposure through microservice apis. In *European Conference on Software Architecture*, pages 3–18. Springer, 2022.

[145] G. Georgakoudis, K. Parasyris, and D. Beckingsale. Proteus: Portable runtime optimization of gpu kernel execution with just-in-time compilation. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 507–522, 2025.

[146] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. Maguire Jr, and D. Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 807–827, 2022.

[147] H. Ghasemirahni, A. Farshin, D. Kostic, and M. Chiesa. Just-in-time packet state prefetching. *arXiv preprint arXiv:2407.04344*, 2024.

[148] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

[149] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 150–160, 2020.

[150] S. Ghosh, M. Shetty, C. Bansal, and S. Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.

[151] A. Gluck. Introducing domain-oriented microservice architecture. Accessed June 2024 from https://www.uber.com/blog/microservice-architecture/, 2020.

[152] Go. Go testing framework. Accessed from https://pkg.go.dev/testing.

[153] J. Goergen, V. Anand, H. Li, J. Li, and A. Kaufmann. Columbo: Low level end-to-end system traces through modular full-system simulation. *arXiv preprint arXiv:2408.05251*, 2024.

[154] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian. A holistic machine learning-based autoscaling approach for microservice applications. In *CLOSER*, pages 190–198, 2021.

[155] GoogleCloudPlatform. Online boutique, fka hipstershop. https://github.com/GoogleCloudPlatform/microservices-demo.

[156] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach. Befaas: An application-centric benchmarking framework for faas platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–8. IEEE, 2021.

[157] M. Grambow, E. Wittern, and D. Bermbach. Benchmarking the performance of microservice applications. *ACM SIGAPP Applied Computing Review*, 20(3):20–34, 2020.

[158] D. Granata, M. Rak, and G. Salzillo. Metasend: A security enabled development life cycle meta-model. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[159] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302. IEEE, 2017.

[160] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53. IEEE, 2017.

[161] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in dyc. *ACM SIGPLAN Notices*, 34(5):293–304, 1999.

[162] J. Grohmann, S. Eismann, S. Elflein, J. v. Kistowski, S. Kounev, and M. Mazkatli. Detecting parametric dependencies for performance models using feature selection techniques. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 309–322. IEEE, 2019.

[163] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones. Monitorless: Predicting performance degradation in cloud applications with machine learning. In *Proceedings of the 20th international middleware conference*, pages 149–162, 2019.

[164] J. Grohmann, M. Straesser, A. Chalbani, S. Eismann, Y. Arian, N. Herbst, N. Peretz, and S. Kounev. Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 165–176, 2021.

[165] Z. Guan, J. Lin, and P. Chen. On anomaly detection and root cause analysis of microservice systems. In *International Conference on Service-Oriented Computing*, pages 465–469. Springer, 2018.

[166] C. Guerrero, I. Lera, and C. Juiz. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*, 16(1):113–135, 2018.

[167] C. Guerrero, I. Lera, and C. Juiz. Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. *Future Generation Computer Systems*, 97:131–144, 2019.

[168] C. Guerrero, I. Lera, and C. Juiz. A lightweight decentralized service placement policy for performance optimization in fog computing. *Journal of Ambient Intelligence and Humanized Computing*, 10(6):2435–2452, 2019.

[169] A. Guerriero and L. Giamattei. Microsertest: a flexible tool for automated testing of microservices applications.

[170] D. Guhathakurta, P. Aggarwal, S. Nagar, R. Arora, and B. Zhou. Utilizing persistence for post facto suppression of invalid anomalies using system logs. 2022.

[171] E. Haddad. Service-oriented architecture: Scaling the uber engineering codebase as we grow. (September 2015). Retrieved October 2020 from https://eng.uber.com/service-oriented-architecture/, 2015.

[172] A. Hakamian. Engineering resilience: Predicting the change impact on performance and availability of reconfigurable systems. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 143–146. IEEE, 2020.

[173] M. Hashemi. (January 2017). Retrieved February 2021 from https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html, 2017.

[174] W. Hasselbring. Benchmarking as empirical standard in software engineering research. In *Evaluation and Assessment in Software Engineering*, pages 365–372. 2021.

[175] W. He, H. Yu, L. Wang, and T. Oh. Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 322–333. IEEE, 2024.

[176] Z. He, P. Chen, X. Li, Y. Wang, G. Yu, C. Chen, X. Li, and Z. Zheng. A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[177] G. Heiser. System benchmarking crimes. https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html.

[178] Hilbert-Yaa. Deathstarbench local tracing. Retrieved August 2022 from https://github.com/Hilbert-Yaa/DeathStarBench/tree/proto-tracing, 2021.

[179] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, 1994.

[180] A. Horn, H. M. Fard, and F. Wolf. Multi-objective hybrid autoscaling of microservices in kubernetes clusters. In *European Conference on Parallel Processing*, pages 233–250. Springer, 2022.

[181] M. R. Hossen and M. A. Islam. Practical efficient microservice autoscaling.

[182] M. R. Hossen and M. A. Islam. Towards efficient microservices management through opportunistic resource reduction.

[183] M. R. Hossen and M. A. Islam. A lightweight workload-aware microservices autoscaling with qos assurance. *arXiv preprint arXiv:2202.00057*, 2022.

[184] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo. Ant-man: towards agile power management in the microservice era. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1098–1111. IEEE Computer Society, 2020.

[185] X. Hou, J. Liu, C. Li, and M. Guo. Unleashing the scalability potential of power-constrained data center in the microservice era. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[186] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[187] L. Huang and T. Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021.

[188] D. Huye, Y. Shkuro, and R. R. Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[189] L. Iffländer, N. Rawtani, L. Beierlieb, N. Fella, K.-D. Lange, and S. Kounev. Attack-aware security function chain reordering. *arXiv preprint arXiv:2005.08364*, 2020.

[190] T. Inagaki, Y. Ueda, T. Nakaike, and M. Ohara. Profile-based detection of layered bottlenecks. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 197–208, 2019.

[191] T. Inagaki, Y. Ueda, M. Ohara, S. Choochotkaew, M. Amaral, S. Trent, T. Chiba, and Q. Zhang. Detecting layered bottlenecks in microservices. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 385–396. IEEE, 2022.

[192] R. Isaacs, P. Alvaro, R. Majumdar, K.-K. Muniswamy-Reddy, M. Salamati, and S. Soudjani. Analyzing metastable failures. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 172–178, 2025.

[193] M. R. Islam, A. Al Maruf, and T. Cerny. Code smell prioritization with business process mining and static code analysis: A case study. *Electronics*, 11(12):1880, 2022.

[194] A. A. T. Isstaif. Self-managed services using mirageos unikernels. In *Proceedings of the 21st International Middleware Conference Doctoral Symposium*, pages 35–37, 2020.

[195] istio. Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/.

[196] ivanium. Deathstarbench social network no tracing. Retrieved August 2022 from https://github.com/ivanium/DeathStarBench/commit/01360df6653e12982335838c877cc4178a518987, 2021.

[197] J. Ivers, C. Seifried, and I. Ozkaya. Untangling the knot: Enabling architecture evolution with search-based refactoring. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 101–111. IEEE, 2022.

[198] R. Iyer, K. Argyraki, and G. Candea. Performance interface extractor (pix) artifact. Accessed November 2024 from https://github.com/dslab-epfl/pix, 2022.

[199] R. Iyer, K. Argyraki, and G. Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, 2022.

[200] S. Jacob, Y. Qiao, and B. Lee. Detecting cyber security attacks against a microservices application using distributed tracing. In *ICISSP*, pages 588–595, 2021.

[201] S. Jacob, Y. Qiao, Y. Ye, and B. Lee. Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks. *Computers & Security*, 118:102728, 2022.

[202] E. Jain and S. Roy. Phase directed compiler optimizations. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 270–279. IEEE, 2016.

[203] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[204] jhipster. Jhipster. Retrieved August 2022 from https://github.com/jhipster/generator-jhipster, 2019.

[205] S. Ji, W. Wu, and Y. Pu. Multi-indicators prediction in microservice using granger causality test and attention lstm. In *2020 IEEE World Congress on Services (SERVICES)*, pages 77–82. IEEE, 2020.

[206] T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang, and Y. Liu. Machine deserves better logging: a log enhancement approach for automatic fault diagnosis. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 106–111. IEEE, 2018.

[207] Z. Jia and E. Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices extended abstract.

[208] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu. Lessons learned from migrating complex stateful applications onto serverless platforms. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 89–96, 2021.

[209] D. Johansson. Model-driven development for microservices: A domain-specific modeling language for kubernetes, 2022.

[210] C. T. Joseph and K. Chandrasekaran. Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments. *Journal of Systems Architecture*, 111:101785, 2020.

[211] A. Jung, C. Crăciunoiu, N. Karaolidis, H. Lefeuvre, D. O. Rubio, F. Huici, C. Rotsos, and P. Olivier. Wayfinder: Automated operating system specialization. *arXiv preprint arXiv:2603.23425*, 2026.

[212] H. Kadiyala, A. Misail, and J. Rubin. Kuber: cost-efficient microservice deployment planner. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 252–262. IEEE, 2022.

[213] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[214] M. Kambadur and M. A. Kim. Nrg-loops: adjusting power from within applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 206–215, 2016.

[215] M. Kang and J. Kim. Comparison of service description and composition for complex 3-tier cloud-based services. *Proceedings of the Asia-Pacific Advanced Network*, 44:37–40, 2017.

[216] M. Kang, J.-S. Shin, and J. Kim. Protected coordination of service mesh for container-based 3-tier service traffic. In *2019 International Conference on Information Networking (ICOIN)*, pages 427–429. IEEE, 2019.

[217] G. M. Karagiannis and C. Synolakis. Twenty challenges in incident planning. *Journal of Homeland Security and Emergency Management*, 14(2):20160061, 2017.

[218] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth. Automated testing and resilience of microservice's network-link using istio service mesh. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 79–88. IEEE, 2022.

[219] A. Karthikeyan, N. Natarajan, G. Somashekar, L. Zhao, R. Bhagwan, R. Fonseca, T. Racheva, and Y. Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, 2023.

[220] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[221] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolek. Trace link recovery for software architecture documentation. In *European Conference on Software Architecture*, pages 101–116. Springer, 2021.

[222] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. De Lara. {JITServer}: Disaggregated caching {JIT} compiler for the {JVM} in the cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, 2022.

[223] E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 251–263, New York, NY, USA, 2002. Association for Computing Machinery.

[224] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[225] P. Kokić. *Unaprijedjenje sigurnosti pomoću mikroservisa*. PhD thesis, University of Zagreb. Faculty of Organization and Informatics. Department of ..., 2018.

[226] J. Kosińska and K. Zieliński. Autonomic management framework for cloud-native applications. *Journal of Grid Computing*, 18(4):779–796, 2020.

[227] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, et al. Apiary: A dbms-backed transactional function-as-a-service framework. *arXiv preprint arXiv:2208.13068*, 2022.

[228] N. Kratzke. About the complexity to transfer cloud applications at runtime and how container platforms can contribute? In *International Conference on Cloud Computing and Services Science*, pages 19–45. Springer, 2017.

[229] N. Kratzke. Smuggling multi-cloud support into cloud-native applications using elastic container platforms. In *CLOSER*, pages 29–42, 2017.

[230] N. Kratzke. About an immune system understanding for cloud-native applications. *CLOUD COMPUTING 2018*, page 41, 2018.

[231] N. Kratzke. About being the tortoise or the hare?-a position paper on making cloud applications too fast and furious for attackers. *arXiv preprint arXiv:1802.03565*, 2018.

[232] N. Kratzke and P.-C. Quint. Project cloudtransit.

[233] R. Laigner, Y. Zhou, and M. A. V. Salles. A distributed database system for event-based microservices. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 25–30, 2021.

[234] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[235] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[236] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[237] I.-T. A. Lee, Z. Zhang, A. Parwal, and M. Chabbi. The tale of errors in microservices. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–36, 2024.

[238] W.-C. Lee, Y. Liu, P. Liu, S. Ma, H. Choi, X. Zhang, and R. Gupta. White-box program tuning. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 122–135. IEEE, 2019.

[239] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. Dominance-based duplication simulation (dbds): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 126–137, 2018.

[240] A. Lesniak, R. Laigner, and Y. Zhou. Enforcing consistency in microservice architectures through event-based constraints. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 180–183, 2021.

[241] J. Levin and T. A. Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[242] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[243] R. Li, M. Du, H. Chang, S. Mukherjee, and E. Eide. Deepstitch: Deep learning for cross-layer stitching in microservices. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, pages 25–30, 2020.

[244] R. Li, M. Du, Z. Wang, H. Chang, S. Mukherjee, and E. Eide. Longtale: Toward automatic performance anomaly explanation in microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 5–16, 2022.

[245] X. Li, Y. Chen, and Z. Lin. Towards automated inter-service authorization for microservice applications. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 3–5, 2019.

[246] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3971–3988, 2021.

[247] Y. Li, T. Li, P. Shen, L. Hao, W. Liu, S. Wang, Y. Song, and L. Bao. Sim-drs: a similarity-based dynamic resource scheduling algorithm for microservice-based web systems. *PeerJ Computer Science*, 7:e824, 2021.

[248] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, et al. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.

[249] Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui, et al. Actionable and interpretable fault localization for recurring failures in online service systems. *arXiv preprint arXiv:2207.09021*, 2022.

[250] H. Lim, Y. Kim, and K. Sun. Service management in virtual machine and container mixed environment using service mesh. In *2021 International Conference on Information Networking (ICOIN)*, pages 528–530. IEEE, 2021.

[251] C. Lima, J. Cezar, G. V. Leobas, E. Rohou, and F. M. Q. Pereira. Guided just-in-time specialization. *Science of Computer Programming*, 185:102318, 2020.

[252] F. Liu, Y. Wang, Z. Li, R. Ren, H. Guan, X. Yu, X. Chen, and G. Xie. Microcbr: Case-based reasoning on spatio-temporal fault knowledge graph for microservices troubleshooting. In *International Conference on Case-Based Reasoning*, pages 224–239. Springer, 2022.

[253] J. Loff. Deathstarbench inconsistency. Retrieved August 2022 from https://github.com/jfloff/antipode-deathstarbench, 2022.

[254] J. Loff, D. Porto, C. Baquero, J. Garcia, N. Preguiça, and R. Rodrigues. Transparent cross-system consistency. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2017.

[255] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei. Fitness-guided resilience testing of microservice-based applications. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 151–158. IEEE, 2020.

[256] N. D. S. Loureiro. *Compositional analysis of vulnerabilities in microservice-based applications*. PhD thesis, 2021.

[257] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[258] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[259] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, and C. Lin. Microservice deployment in edge computing based on deep q learning. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[260] T. Ma*, Y. Chen*, V. Anand, A. Cornacchia, A. Faustino, G. Liu, S. Zhang, H. Luo, S. Fahmy, Z. Qazi, and M. Canini. Maestro: Multi-agent evaluation suite for testing, reliability, and observability. *arXiv preprint arxiv:2601.00481*, 2026.

[261] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 589–603, 2015.

[262] J. Mace and R. Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–18, 2018.

[263] N. Marie-Magdelaine and T. Ahmed. Proactive autoscaling for cloud-native applications using machine learning. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–7. IEEE, 2020.

[264] D. R. Mathews, M. Verma, J. Lakshmi, and P. Aggarwal. Towards more effective and explainable fault management using cross-layer service topology. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 94–96. IEEE, 2022.

[265] A. P. Matthews. *Microservice Pattern Identification from Recovered Architectures of Orchestrated Systems*. University of California, Irvine, 2021.

[266] M. Mazkatli, D. Monschein, J. Grohmann, and A. Koziolek. Incremental calibration of architectural performance models with parametric dependencies. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 23–34. IEEE, 2020.

[267] J. Mccandless and D. Gregg. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–20, 2012.

[268] A. Mcdade. Significant outage for amazon web services stalls netflix, delta airlines, others. Retrieved September 2022 from https://www.newsweek.com/significant-outage-amazon-web-services-stalls-netflix-delta-airlines-others-1657077, 2021.

[269] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão. Data replication on the cloud/edge. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–7, 2019.

[270] P. Mehrotra*, V. Anand*, D. Margo, M. Rezaei, and M. Seltzer. Sok: The faults in our graph benchmarks. *arXiv preprint arXiv:2404.00766*, 2024.

[271] W. D. Mendonça, W. K. Assunção, L. V. Estanislau, S. R. Vergilio, and A. Garcia. Towards a microservices-based product line with multi-objective evolutionary algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[272] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1148–1164, 2022.

[273] microservices demo. Retrieved August 2022 from https://github.com/microservices-demo/microservices-demo, 2016.

[274] A. Mirhosseini, S. Elnikety, and T. F. Wenisch. Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 442–457, 2021.

[275] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198. IEEE, 2019.

[276] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi. Phase-aware optimization in approximate computing. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 185–196. IEEE, 2017.

[277] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[278] H. Mohamed and O. El-Gayar. End-to-end latency prediction of microservices workflow on kubernetes: A comparative evaluation of machine learning models and resource metrics. In *Proceedings of the 54th Hawaii International Conference on System Sciences*, page 1717, 2021.

[279] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári. Dataplane specialization for high-performance openflow software switching. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 539–552, 2016.

[280] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[281] J. Mukherjee, A. Baluta, M. Litoiu, and D. Krishnamurthy. Rad: Detecting performance anomalies in cloud-based web services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 493–501. IEEE, 2020.

[282] V. I. Munteanu, A. Edmonds, T. M. Bohnert, and T.-F. Fortis. Cloud incident management, challenges, research directions, and architectural approach. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 786–791. IEEE, 2014.

[283] G. Muntoni, J. Soldani, and A. Brogi. Mining the architecture of microservice-based applications from their kubernetes deployment. In *European Conference on Service-Oriented and Cloud Computing*, pages 103–115. Springer, 2020.

[284] A. Nadeem and M. Z. Malik. A case for microservices orchestration using workflow engines. *arXiv preprint arXiv:2204.07210*, 2022.

[285] S. Nagar, S. Samanta, P. Mohapatra, and D. Kar. Building golden signal based signatures for log anomaly detection. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 203–208. IEEE, 2022.

[286] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin. {BASTION}: A security enforcement network stack for container networks. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 81–95, 2020.

[287] A. R. Nasab, M. Shahin, S. A. H. Raviz, P. Liang, A. Mashmool, and V. Lenarduzzi. An empirical study of security practices for microservices systems. *arXiv preprint arXiv:2112.14927*, 2021.

[288] S. Naser Moghaddasi, H. Smajlović, A. Shajii, and I. Numanagić. Vectron: A dynamic programming auto-vectorization framework. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 644–659, 2025.

[289] T. Nebaeus. Optimal scaling configurations for microservice-oriented architectures using genetic algorithms, 2019.

[290] S. Needleman. Amazon outage disrupts lives, surprising people about their cloud dependency. Retrieved September 2022 from https://www.wsj.com/articles/amazon-outage-disrupts-lives-surprising-people-about-their-cloud-dependency-11638972001, 2021.

[291] C. Nguyen, A. Mehta, C. Klein, and E. Elmroth. Why cloud applications are not ready for the edge (yet). In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 250–263, 2019.

[292] H. X. Nguyen, S. Zhu, and M. Liu. Graph-phpa: Graph-based proactive horizontal pod autoscaling for microservices using lstm-gnn. *arXiv preprint arXiv:2209.02551*, 2022.

[293] S. Nijboer. Traffic analysis of a service mesh. Master's thesis, 2021.

[294] L. Nolan. Why health checks are like sidewalks. (December 2022). Retrieved April 2023 from https://www.usenix.org/publications/loginonline/why-health-check-sidewalk, 2022.

[295] E. T. Nordli, P. H. Nguyen, F. Chauvel, and H. Song. Event-based customization of multi-tenant saas using microservices. In *International Conference on Coordination Languages and Models*, pages 171–180. Springer, 2020.

[296] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, and W. Hasselbring. Detector-based component model abstraction for microservice-based systems. *Computing*, 103(11):2521–2551, 2021.

[297] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *International Conference on Service-Oriented Computing*, pages 580–596. Springer, 2020.

[298] E. Ntentos, U. Zdun, J. Soldani, and A. Brogi. Assessing architecture conformance to coupling-related infrastructure-as-code best practices: Metrics and case studies. In *European Conference on Software Architecture*, pages 101–116. Springer, 2022.

[299] M. S. Nyfløtt. Optimizing inter-service communication between microservices. Master's thesis, NTNU, 2017.

[300] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, 2022.

[301] A. Osman, J. Born, and T. Strufe. Mitigating internal, stealthy dos attacks in microservice networks. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 500–504. Springer, 2021.

[302] G. Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[303] overleaf. Overleaf. https://github.com/overleaf/overleaf.

[304] H. Pan, J. Dong, M. Xing, and Y. Wu. Synergy-guided compiler auto-tuning of nested llvm pass pipelines. *arXiv preprint arXiv:2510.13184*, 2025.

[305] H. Pan, P. He, Z. Li, P. Zhang, J. Wan, Y. Zhou, X. Duan, Y. Zhang, and G. Xie. Hoda: a high-performance open vswitch dataplane with multiple specialized data paths. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 82–98, 2024.

[306] H. Pan, Y. Wei, M. Xing, Y. Wu, and C. Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 614–627, 2025.

[307] H. Pan, H. Zhang, M. Xing, and Y. Wu. A hybrid, knowledge-guided evolutionary framework for personalized compiler auto-tuning. *arXiv preprint arXiv:2510.14292*, 2025.

[308] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[309] M. Pancholi, A. D. Kellas, V. P. Kemerlis, and S. Sethumadhavan. Timeloops: System call policy learning for containerized microservices. *arXiv preprint arXiv:2204.06131*, 2022.

[310] J. Park, B. Choi, C. Lee, and D. Han. Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[311] S. Park, S. Latifi, Y. Park, A. Behroozi, B. Jeon, and S. Mahlke. Srtuner: Effective compiler optimization customization by exposing synergistic relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 118–130. IEEE, 2022.

[312] M. A. Patterns. Lakeside mutual. https://github.com/Microservice-API-Patterns/LakesideMutual.

[313] K. Perera and I. Perera. A rule-based system for automated generation of serverless-microservices architecture. In *2018 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–8. IEEE, 2018.

[314] H. Phalachandra, P. Bhatt, I. Agarwal, and R. Bhowmick. Improving task scheduling in microservice environments by considering intra-job dependencies. In *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 568–573. IEEE, 2022.

[315] pktgen. Pktgen – dpdk traffice generator. https://github.com/pktgen/Pktgen-DPDK. Retrieved Mar 30, 2026.

[316] G. Poesia and F. M. Q. Pereira. Dynamic dispatch of context-sensitive optimizations. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[317] M. Popov, C. Akel, Y. Chatelain, W. Jalby, and P. de Oliveira Castro. Piecewise holistic autotuning of parallel programs with cere. *Concurrency and Computation: Practice and Experience*, 29(15):e4190, 2017.

[318] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1216, 2020.

[319] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2021.

[320] D. R. Prasanna. Dependency injection. 2009.

[321] A. Prokopec, G. Duboscq, D. Leopoldseder, and T. Würthinger. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 164–179. IEEE, 2019.

[322] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 805–825, 2020.

[323] P.-C. Quint and N. Kratzke. Towards a description of elastic cloud-native applications for transferable multi-cloud-deployments. *Small*, 147:34, 2017.

[324] P.-C. Quint and N. Kratzke. Towards a lightweight multi-cloud dsl for elastic and transferable cloud-native applications. *arXiv preprint arXiv:1802.03562*, 2018.

[325] F. Rademacher, S. Sachweh, and A. Zündorf. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 IEEE International conference on software architecture (ICSA)*, pages 21–30. IEEE, 2019.

[326] F. Rademacher, S. Sachweh, and A. Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020.

[327] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf. Towards a viewpoint-specific metamodel for model-driven development of microservice architecture. *arXiv preprint arXiv:1804.09948*, 2018.

[328] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf. Specific model-driven microservice development with interlinked modeling languages. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 57–5709. IEEE, 2019.

[329] F. Rademacher, J. Sorgalla, P. N. Wizenty, S. Sachweh, and A. Zündorf. Microservice architecture and model-driven development: Yet singles, soon married (?). In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–5, 2018.

[330] J. Rahman and P. Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.

[331] M. I. Rahman, S. Panichella, and D. Taibi. A curated dataset of microservices-based systems, extended version. Vol-2520, 2019.

[332] K. Ramasubramanian, A. Raina, J. Mace, and P. Alvaro. Act now: Aggregate comparison of traces for incident localization. *arXiv preprint arXiv:2205.06933*, 2022.

[333] V. Rao, V. Singh, K. Goutham, B. U. Kempaiah, R. J. Mampilli, S. Kalambur, and D. Sitaram. Scheduling microservice containers on large core machines through placement and coalescing. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 80–100. Springer, 2021.

[334] M. Rasiuckevicius. Longest prefix match (lpm) library. Accessed June 2025 from https://github.com/rmind/liblpm, 2022.

[335] K. Ray and A. Banerjee. Horizontal auto-scaling for multi-access edge computing using safe reinforcement learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(6):1–33, 2021.

[336] redis. Redis patterns example. https://redis.io/docs/reference/patterns/twitter-clone/.

[337] J. S. Rellermeyer, M. Amer, R. Smutzer, and K. Rajamani. Container density improvements with dynamic memory extension using nand flash. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.

[338] F. G. O. Risso and S. Galantino. Enabling job aware scheduling on kubernetes cluster.

[339] rnp68. Teastore cve-2021-44228. Retrieved September 2022 from https://github.com/rnp68/TeaStore, 2021.

[340] O. R. C. Rodríguez, C. Pahl, N. El Ioini, H. R. Barzegar, et al. Improvement of edge computing workload placement using multi objective particle swarm optimization. In *2021 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 1–8. IEEE, 2021.

[341] Y. Rouf, J. Mukherjee, M. Litoiu, J. Wigglesworth, and R. Mateescu. A framework for developing devops operation automation in clouds using components-off-the-shelf. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 265–276, 2021.

[342] P. Roy and X. Liu. Structslim: A lightweight profiler to guide structure splitting. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 36–46, 2016.

[343] F. Ruffy, Z. Wang, G. Antichi, A. Panda, and A. Sivaraman. Incremental specialization of network programs. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 264–272, 2024.

[344] W. Runan, G. Casale, and A. Filieri. Service distribution estimation for microservices using markovian arrival processes.

[345] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the 16th international conference on Supercomputing*, pages 274–284, 2002.

[346] E. Rwemigabo. *Autonomic application topology redistribution.* PhD thesis, 2018.

[347] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[348] V. Sachidananda and A. Sivaraman. Learned autoscaling for cloud microservices with multi-armed bandits. *arXiv preprint arXiv:2112.14845*, 2021.

[349] M. Sakai and K. Takahashi. Constructing a service process model based on distributed tracing for conformance checking of microservices. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE, 2022.

[350] M. Sakai, K. Takahashi, and S. Kondoh. Method of constructing petri net service model using distributed trace data of microservices. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 214–217. IEEE, 2021.

[351] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 17–27, 2022.

[352] A. R. SAMPAIO JUNIOR. Runtime adaptation of microservices. 2018.

[353] M. Santana, A. Sampaio Jr, M. Andrade, and N. S. Rosa. Transparent tracing of microservice-based applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1252–1259, 2019.

[354] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot. Lukewarm serverless functions: characterization and optimization. In *ISCA*, pages 757–770, 2022.

[355] M. Schiedermeier. A concern-oriented software engineering methodology for micro-service architectures. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, 2020.

[356] M. Schiewe. Bridging the gap between source code and high-level concepts in static code analysis: student research abstract. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1615–1618, 2022.

[357] M. Schiewe, J. Curtis, V. Bushong, and T. Cerny. Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761, 2022.

[358] N. Schmitt, L. Iffländer, A. Bauer, and S. Kounev. Online power consumption estimation for functions in cloud applications. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 63–72. IEEE, 2019.

[359] Y. R. Schneider and A. Koziolek. Towards reverse engineering for component-based systems with domain knowledge of the technologies used.

[360] SEALABQualityGroup. E-shopper. https://github.com/SEALABQualityGroup/E-Shopper.

[361] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan. Pmnet: in-network data persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2021.

[362] M. Selakovic, T. Glaser, and M. Pradel. An actionable performance profiler for optimizing the order of evaluations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 170–180, 2017.

[363] K. Sellami, M. A. Saied, and A. Ouni. A hierarchical dbscan method for extracting microservices from monolithic applications. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, pages 201–210, 2022.

[364] P. V. Seshadri, H. Balagopal, A. Nayak, A. P. Kumar, and P. Loyola. Konveyor move2kube: A framework for automated application replatforming. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 115–124. IEEE, 2022.

[365] V. Seshagiri, S. Balyan, V. Anand, K. Dhole, I. Sharma, A. Wildani, J. Cambronero, and A. Züfle. Chatting with logs: An exploratory study on finetuning llms for logql. *arXiv preprint arXiv:2412.03612*, 2024.

[366] V. Seshagiri, D. Huye, L. Liu, A. Wildani, and R. R. Sambasivan. Sok: Identifying mismatches between. *Journal of Systems Research (JSys) Volume*, 2(1), 2022.

[367] E. Shanks. Kubernetes - desired state and control loops. Accessed July, 2024 from https://theithollow.com/2019/09/16/kubernetes-desired-state-and-control-loops/, 2019.

[368] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[369] R. Sharma and A. Mathur. Traefik as kubernetes ingress. In *Traefik API Gateway for Microservices*, pages 191–245. Springer, 2021.

[370] O. Shchur, A. C. Turkmen, T. Januschowski, J. Gasthaus, and S. Günnemann. Detecting anomalous event sequences with temporal point processes. *Advances in Neural Information Processing Systems*, 34:13419–13431, 2021.

[371] J. Shi, J. Wang, K. Fu, Q. Chen, D. Zeng, and M. Guo. Qos-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 324–334. IEEE, 2022.

[372] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson. {FlightTracker}: Consistency across {Read-Optimized} online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423, 2020.

[373] A. Singjai and U. Zdun. Conformance assessment of architectural design decisions on the mapping of domain model elements to apis and api endpoints. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 76–83. IEEE, 2022.

[374] A. Singjai, U. Zdun, O. Zimmermann, M. Stocker, and C. Pautasso. Patterns on designing api endpoint operations. 2021.

[375] S. Sioutas, S. Stuijk, H. Corporaal, T. Basten, and L. Somers. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 254–264, 2018.

[376] W. E. Smith. Various optimizers for single-stage production. Technical report, 1955.

[377] D. Sokolowski. Deployment coordination for cross-functional devops teams. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1630–1634, 2021.

[378] D. Sokolowski. Infrastructure as code for dynamic deployments. 2022.

[379] J. Soldani and A. Brogi. Automated generation of configurable cloud-native chaos testbeds. In *European Dependable Computing Conference*, pages 101–108. Springer, 2021.

[380] J. Soldani, G. Montesano, and A. Brogi. What went wrong? explaining cascading failures in microservice-based applications. In *Symposium and Summer School on Service-Oriented Computing*, pages 133–153. Springer, 2021.

[381] J. Soldani, G. Muntoni, D. Neri, and A. Brogi. The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621, 2021.

[382] G. Somashekar, A. Dutt, R. Vaddavalli, S. B. Varanasi, and A. Gandhi. B-meg: Bottlenecked-microservices extraction using graph neural networks. 2022.

[383] G. Somashekar and A. Gandhi. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 7–14, 2021.

[384] G. Somashekar, K. Tandon, A. Kini, C.-C. Chang, P. Husak, R. Bhagwan, M. Das, A. Gandhi, and N. Natarajan. OPPerTune: Post-deployment configuration tuning of services made easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1101–1120, 2024.

[385] H. Song, P. H. Nguyen, and F. Chauvel. Using microservices to customize multi-tenant saas: From intrusive to non-intrusive. In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[386] J. Sorgalla. Ajil: A graphical modeling language for the development of microservice architectures. In *Extended Abstracts of the Microservices 2017 Conference*, 2017.

[387] J. Sorgalla, F. Rademacher, S. Sachweh, and A. Zündorf. Model-driven development of microservice architecture: An experiment on the quality in use of a uml-and a dsl-based approach. 2020.

[388] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):1–25, 2021.

[389] S. Speth. Semi-automated cross-component issue management and impact analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1090–1094. IEEE, 2021.

[390] spring.io. Spring boot. Retrieved August 2022 from https://spring.io/projects/spring-boot.

[391] J. Srinivas, W. Ding, and M. Kandemir. Reactive tiling. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 91–102. IEEE, 2015.

[392] A. Sriraman, A. Dhanotia, and T. F. Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[393] A. Sriraman and T. F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[394] A. Sriraman and T. F. Wenisch. μtune: Auto-tuned threading for {OLDI} microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 177–194, 2018.

[395] M. Stephenson and R. Rangan. Pgz: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 36–46, 2021.

[396] A. Suljkanović, B. Milosavljević, V. Indjić, and I. Dejanović. Developing microservice-based applications using the silvera domain-specific language. *Applied Sciences*, 12(13):6679, 2022.

[397] C.-a. Sun, J. Wang, Z. Liu, and Y. Han. A variability-enabling and model-driven approach to adaptive microservice-based systems. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 968–973. IEEE, 2021.

[398] H. Sun, D. Bonetta, C. Humer, and W. Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.

[399] Y. Sun, L. Zhao, Z. Wang, D. Cui, Y. Yang, and Z. Gao. Fault root rank algorithm based on random walk mechanism in fault knowledge graph. In *2021 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6. IEEE, 2021.

[400] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu. Fail through the cracks: Cross-system interaction failures in modern cloud systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 433–451, 2023.

[401] Y. Tang, N. Chen, Y. Zhang, X. Yao, and S. Yu. Fine-grained diagnosis method for microservice faults based on hierarchical correlation analysis. In *CCF Conference on Computer Supported Cooperative Work and Social Computing*, pages 14–28. Springer, 2022.

[402] S. T. Teixeira, C. Ancourt, D. Padua, and W. Gropp. Locus: a system and a language for program optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 217–228. IEEE, 2019.

[403] P. Tennage, S. Perera, M. Jayasinghe, and S. Jayasena. An analysis of holistic tail latency behaviors of java microservices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 697–705. IEEE, 2019.

[404] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, and I. Luković. Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018.

[405] V. Thrivikraman, V. R. Dixit, V. K. Gowda, S. K. Vasudevan, and S. Kalambur. Misertrace: Kernel-level request tracing for microservice visibility. *CoRR*, 2022.

[406] O. Tomarchio, D. Calcaterra, G. Di Modica, and P. Mazzaglia. Torch: a tosca-based orchestrator of multi-cloud containerised applications. *Journal of Grid Computing*, 19(1):1–25, 2021.

[407] K. A. Torkura, C. Meinel, and N. Kratzke. Don't wait to be breached! creating asymmetric uncertainty of cloud applications via moving target defenses. *arXiv preprint arXiv:1901.04319*, 2019.

[408] M. Toslali, E. Ates, A. Ellis, Z. Zhang, D. Huye, L. Liu, S. Puterman, A. K. Coskun, and R. R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with vaif. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–75, 2021.

[409] M. Toslali, S. Parthasarathy, F. Oliveira, H. Huang, and A. K. Coskun. Iter8: Online experimentation in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 289–304, 2021.

[410] L. Traini and V. Cortellessa. Delag: Detecting latency degradation patterns in service-based systems. *arXiv preprint arXiv:2110.11155*, 2021.

[411] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018.

[412] T. Ueda, T. Nakaike, and M. Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016.

[413] T. N. Y. U. T. Ueda and M. Ohara. Performance characteristics of linux for java workloads oversubscribing memory.

[414] M. Ulrich. Addressing cascading failures. Retrieved September 2022 from https://sre.google/sre-book/addressing-cascading-failures/.

[415] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[416] A. van Hoorn and T. F. Düllmann. Continuous testing tool i. 2020.

[417] W. Viktorsson, C. Klein, and J. Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–4. IEEE, 2020.

[418] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[419] H. Vural and M. Koyuncu. Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*, 9:32721–32733, 2021.

[420] A. Walker, D. Das, and T. Cerny. Automated microservice code-smell detection. In *Information Science and Applications*, pages 211–221. Springer, 2021.

[421] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 142–150. IEEE, 2020.

[422] Q. Wang, L. Shwartz, G. Y. Grabarnik, V. Arya, and K. Shanmugam. Detecting causal structure on cloud application microservices using granger causality models. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 558–565. IEEE, 2021.

[423] R. Wang, G. Casale, and A. Filieri. Service demand distribution estimation for microservices using markovian arrival processes. In *International Conference on Quantitative Evaluation of Systems*, pages 310–328. Springer, 2021.

[424] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, and M. Guo. Exploring efficient microservice level parallelism. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[425] S. Waterworth. Stan's robot shop – a sample microservice application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/, 2018.

[426] M. Wawrzoniak, I. Müller, R. Bruno, A. Klimovic, and G. Alonso. Short-lived datacenter. *arXiv preprint arXiv:2202.06646*, 2022.

[427] N. Wenzler. Automated performance regression detection in microservice architectures. B.S. thesis, 2017.

[428] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever. P2go: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 146–152, 2020.

[429] P. Wizenty, J. Sorgalla, F. Rademacher, and S. Sachweh. Magma: Build management-based generation of microservice infrastructures. In *Proceedings of the 11th European conference on software architecture: companion proceedings*, pages 61–65, 2017.

[430] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[431] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.

[432] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao. Performance diagnosis in cloud microservices using deep learning. In *AIOPS 2020-International Workshop on Artificial Intelligence for IT Operations*, 2020.

[433] L. Wu, J. Tordsson, A. Acker, and O. Kao. Microras: Automatic recovery in the absence of historical failure data for microservice systems. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 227–236. IEEE, 2020.

[434] L. Wu, J. Tordsson, E. Elmroth, and O. Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[435] S. Wu, Q. Su, Z. Niu, and H. Xu. Tomur: Traffic-aware performance prediction of on-nic network functions with multi-resource contention. *arXiv preprint arXiv:2405.05529*, 2024.

[436] X. Wu, S. Cheng, S. Yuan, and Z. Wang. A parallelism-based earliest finish time (pbeft) algorithm for workflow scheduling in clouds. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 22–26. IEEE, 2022.

[437] Y. Xia. *Reducing the Length of Field-replay Based Load Testing*. PhD thesis, Concordia University, 2021.

[438] xiling42. Zl-ldfi with train ticket. Retrieved September 2022 from https://github.com/xiling42/train-ticket-LDFI, 2020.

[439] R. Xin, P. Chen, and Z. Zhao. Causalrca: Causal inference based precise fine-grained root cause localization for microservice applications. *arXiv preprint arXiv:2209.02500*, 2022.

[440] J. Xu, P. Chen, L. Yang, F. Meng, and P. Wang. Logdc: Problem diagnosis for declartively-deployed cloud applications with log. In *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, pages 282–287. IEEE, 2017.

[441] M. Xu and R. Buyya. Brownoutcon: A software system based on brownout and containers for energy-efficient cloud computing. *Journal of Systems and Software*, 155:91–103, 2019.

[442] M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):667–697, 2002.

[443] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic. Lessons learned from hyper-parameter tuning for microservice candidate identification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1141–1145. IEEE, 2021.

[444] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic. Partitioning cloud-based microservices (via deep learning). *arXiv preprint arXiv:2109.14569*, 2021.

[445] K. Yin and Q. Du. On representing resilience requirements of microservice architecture systems. *International Journal of Software Engineering and Knowledge Engineering*, 31(06):863–888, 2021.

[446] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, pages 3087–3098, 2021.

[447] G. Yu, P. Chen, and Z. Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75. IEEE, 2019.

[448] G. Yu, P. Chen, and Z. Zheng. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Transactions on Cloud Computing*, 2020.

[449] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. The shift from models to compound ai systems. Accessed July 2025 from https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[450] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 275–290, 2019.

[451] U. Zdun, P.-J. Queval, G. Simhandl, R. Scandariato, S. Chakravarty, M. Jelic, and A. Jovanovic. Microservice security metrics for secure communication, identity management, and observability. *ACM Transactions on Software Engineering and Methodology*, 2022.

[452] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. 2022.

[453] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1187–1204, 2020.

[454] K. Zhang, A. Filieri, and R. Chatley. Kubedim: Self-adaptive service degradation of microservices-based systems. 2021.

[455] L. Zhang, V. Anand, Z. Xie, Y. Vigfusson, and J. Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. *arXiv preprint arXiv:2202.05769*, 2022.

[456] L. Zhang, Z. Xie, V. Anand, Y. Vigfusson, and J. Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023.

[457] Y. Zhang, Y. Gan, and C. Delimitrou. μqsim: Enabling accurate and scalable simulation for interactive microservices. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 212–222. IEEE, 2019.

[458] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou. Sinan: Ml-based & qos-aware resource management for cloud microservices.

[459] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou. 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*, pages 167–181, 2021.

[460] Y. Zhang, N. Sobotka, S. Park, S. Jamilan, T. A. Khan, B. Kasikci, G. A. Pokam, H. Litz, and J. Devietti. Rpg2: Robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 999–1013, 2024.

[461] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[462] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[463] P. Zhao and J. N. Amaral. Feedback-directed switch-case statement optimization. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 295–302. IEEE, 2005.

[464] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.

[465] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*, 2019.

[466] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 683–694, 2019.

[467] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding. Delta debugging microservice systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 802–807. IEEE, 2018.

[468] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Poster: Benchmarking microservice systems for software engineering research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 323–324. IEEE, 2018.

[469] L. Zhu. Decomposition tool i. 2019.

[470] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, et al. Dissecting service mesh overheads. *arXiv preprint arXiv:2207.00592*, 2022.

[471] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker. Interface responsibility patterns: processing resources and operation responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–24, 2020.

[472] O. Zimmermann, C. Pautasso, D. Lübke, U. Zdun, and M. Stocker. Data-oriented interface responsibility patterns: Types of information holder resources. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–25, 2020.

[473] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun. Introduction to microservice api patterns (map). 2019.

[474] W. Zorgdrager, K. Psarakis, M. Fragkoulis, E. Visser, and A. Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. *arXiv preprint arXiv:2112.00710*, 2021.

Part V

Appendix

Appendix A

Plugins for Executing Blueprint Applications

In this appendix, we provide additional implementation details for Blueprint's testing and workload-generation plugins, which provide execution support for generated applications. We provide annotated code examples to show how users can utilize these plugins for automatically generating blackbox tests and workload generators compatible with the generated implementation.

A.1 Blueprint's Testing Plugin

Tests are important for developers for ensuring that the application is functionally correct. Typically, developers use tests both during development-time and during live testing of the real system. For development-time tests, developers typically write unit tests as well as integration tests. For live tests, developers

```
var userServiceRegistry = registry.NewServiceRegistry[socialnetwork.UserService]("user_service")
var userCacheRegistry = registry.NewServiceRegistry[backend.Cache]("user_cache")
var userDBRegistry = registry.NewServiceRegistry[backend.NoSQLDatabase]("user_db")

// Optional Registry initialization for local test execution
func init() {
    userCacheRegistry.Register("local", func(ctx context.Context) (backend.Cache, error) {
        return simplecache.NewSimpleCache(ctx)
    })
    userDBRegistry.Register("local", func(ctx context.Context) (backend.NoSQLDatabase, error) {
        return simplenosqldb.NewSimpleNoSQLDB(ctx)
    })
    userServiceRegistry.Register("local", func(ctx context.Context) (socialnetwork.UserService, error) {
        cache, _ := userCacheRegistry.Get(ctx)
        db, _ := userDBRegistry.Get(ctx)
        return socialnetwork.NewUserServiceImpl(ctx, cache, db)
    })
}

func TestRegisterUserWithId(t *testing.T) {
    ctx := context.Background()
    service, err := userServiceRegistry.Get(ctx)
    require.NoError(t, err)

    // Test User Registration
    user := socialnetwork.User{FirstName:  "Thesis", LastName:  "Reader", Username:  "tr", UserID: 5}
    err = service.RegisterUserWithId(ctx, user, "password")
    require.NoError(t, err)
}
```

(a) Test implementation as part of the workflow specification

```
func init() {
    // Initialize the client library process to define all the objects for service clients.
    clientlib := clients.NewClientLibrary("tests")
    userServiceRegistry.Register("user_service", func(ctx context.Context) (socialnetwork.UserService,
     error) {
        // Build the client library
        namespace, err := clientlib.Build(ctx)
        if err != nil {
            return nil, err
        }

        // Get and return the client
        var client socialnetwork.UserService
        err = namespace.Get("user_service.client", &client)
        return client, err
    })
    userServiceRegistry.SetDefault("user_service")
    // Client registrations for other registries
    ...
}
```

(b) Blueprint generated code for supporting variant specific tests

Fig. A.1: Sample End-to-End Blackbox test for UserService from the Social Network application

typically rely upon ad-hoc scripts that execute the ingress API calls in a black-box fashion, *i.e.* only ensuring the correctness of the output of the calls without checking the internal states of the system.

A.1.1 Go Testing Framework

To support both these types of tests, Blueprint relies on Go's standardized testing framework [152]. This framework provides support for automated testing of Go code. To write tests with this framework, developers implement the test as functions of the form "`func TestXxx(*testing.T)`" that are placed in source code files with names that end with "`_test.go`". Tests use "`require`" and "`assert`" statements to check for desired correctness. To execute these tests, developers use the built-in "`go test`" command that executes all tests in the folder where the command is executed.

A.1.2 Workflow-based Testing

Blueprint's `gotests` plugin uses Go's testing framework to allow developers to implement reusable end-to-end tests. With the plugin, developers implement tests the same way they would if they were writing tests for regular Go code. Each test lives in a function of the form "func TestXxx(*testing.T)" and must be placed in source code files with names that end with "`_test.go`" as part of the workflow specification of the application. Blueprint then generates the relevant glue code to tie the test implementation from the workflow to the generated implementation of the system. Developers can then execute the tests against the deployed system with the built-in "`go test`" command. To ensure that Blueprint

can correctly generate glue code, the plugin places a restriction on the test implementation: the test must not directly instantiate service objects. To do so, the plugin provides the concept of service registries.

Service Registry. Developers use service registries to instantiate and provide objects for the specific service under test. A service registry is responsible for generating objects that are compatible with the generated system implementation. Service registry provides a thin interface consisting of a `Register` function to register build functions for instantiating objects of a service and a `Get` function that executes the registered function to return the instantiated object.

Fig. A.1 shows the use of service registries in test implementation. First, developers declare a service registry for each service they wish to test (Lines 1-3, Fig. A.1a). Next, developers use the registry to get an instance of the object (Line 22, Fig. A.1a), which developers then use to make calls on the instantiated object and check for correctness (Lines 26-28, Fig. A.1a). To ensure that the service registry returns an object that is compatible with the generated implementation of the system, Blueprint's `gotests` generates glue code for building compatible objects, as shown in Fig. A.1b. Here, the plugin generates code that registers a build function for building an object of the specific service under test using a runtime namespace that contains the client objects of that service (Lines 4-15, Fig. A.1b). As this code is inside Go's `init` function, it executes before any tests and ensures that the Tests will use the correct client objects. The init function along with the client libraries are re-generated whenever the developer uses Blueprint to regenerate a new implementation of the system.

Service registries are also compatible with executing tests against just the workflow specification during development time. To do so, developers must register build functions that instantiate the service object directly from the workflow specification. Lines 7-17 in Fig. A.1a show how local registries can generate service objects for test executions during development time.

Test Semantics. Typically, Go's testing framework executes the tests and restores the state to allow for continuous development with test execution. However, tests executed against a deployed system will modify the state of the system without resetting the state of the system. As a result, these tests are only guaranteed to be safely executed exactly once. Note that the test itself may clean up the state of the system and revert the changes it has made. This would allow black-box tests to be executed more than once against the deployed system. However, this is neither a requirement for Blueprint's testing plugin nor is it enforced.

A.2 Blueprint's Workload Plugin

Blueprint provides a `workload` plugin to support the generation of processes that exercise a programmatically defined workload against a Blueprint-generated deployed application. Similar to Blueprint's `gotest` plugin, Blueprint's `workload` plugin also removes the manual effort necessary for reimplementing workload processes for different system implementations.

```
1  // RequestFunction wraps around the request execution and response processing.
2  type RequestFunction func(context.Context) Stat
3
4  // APIInfo encapsulates the execution information for each API.
5  type ApiInfo struct {
6      Name       string
7      Fn         RequestFunction
8      Proportion int
9  }
10
11 // Workload that is to be executed against the target application.
12 type Workload struct {
13     ApiInfos []ApiInfo
14 }
15
16 func (w *Workload) AddAPI(name string, fn RequestFunction, proportion int) error {
17     info := ApiInfo{Name: name, Fn: fn, Proportion: proportion}
18     w.ApiInfos = append(w.ApiInfos, info)
19     return nil
20 }
21
22 // Engine for executing the Workload
23 type Engine struct {...}
24 func NewEngine(stats_filename string, request_rate int64, duration string, workload * Workload) {...}
25 func (e * Engine) RunOpenLoop(ctx context.Context) {...}
```

Fig. A.2: APIs exported by Blueprint's `workload` plugin

A.2.1 Plugin APIs

The `workload` plugin provides APIs and type definitions (shown in Fig. A.2). The plugin decomposes the workload into two orthogonal components - (i) the workload definition and (ii) the workload execution.

Defining the Workload. The workload definitions (Lines 12-20) consists of a list of APIs that will be executed. To add an API (Lines 5-9) to a workload, developers must provide three key pieces of information: (i) the name of the API; (ii) an execution function of type `RequestFunction` (Line 2) that executes the request, processes the response, and returns request attributes and stats; and (iii) the percentage of the workload that must correspond to the requests for this

```go
type ComplexWldGen struct {
    compose_post ComposePostService
}

func NewComplexWldGen(ctx context.Context, compose_post ComposePostService) (*ComplexWldGen, error) {
    return &ComplexWldGen{compose_post}, nil
}

// User-defined function for generating request arguments for the ComposePost API
func GenComposePostArgs() (int64, string) {...}

// Helper wrapper function for executing requests and capturing statistics
func statWrapper(fn func() error) workload.Stat {
    start := time.Now()
    err := fn()
    duration := time.Since(start)
    s := workload.Stat{}
    s.Start = start.UnixNano()
    s.Duration = duration.Nanoseconds()
    s.IsError = (err != nil)
    return s
}

// RequestFunction for executing a request for the ComposePost API
func (w *ComplexWldGen) RunComposePost(ctx context.Context) workload.Stat {
    userID, postContent := GenComposePostArgs()
    return statWrapper(func() error {
        _, err := w.compose_post.ComposePost(ctx, userID, postContent)
        return err
    })
}

// Implements the Runnable interface provided by the runtime namespace
func (w *ComplexWldGen) Run(ctx context.Context) error {
    wrk := workload.NewWorkload()
    wrk.AddAPI("ComposePost", w.RunComposePost, 100)
    engine, _ := workload.NewEngine("stats.csv", 1000 /* requests/s */, "10s" /* duration */, wrk)
    engine.RunOpenLoop(ctx)
    return engine.PrintStats()
}
```

Fig. A.3: Sample usage of the workload plugin for the Social Network application in its workflow spec

API. For a workload to be valid, the percentages for all API must cumulatively sum to 100.

Executing the Workload. To execute the workload, the `workload` plugin provides an engine (Lines 23-25) that executes the defined workload. The engine currently exports two configuration knobs that developers can use to modify the execution behaviors - (i) the request rate controls the number of requests to be submitted per second (the requests are generated with a poisson distribution in an open-loop fashion); and (ii) the workload duration which controls how long the requests should be generated. The engine can be extended to support different request distributions. To support developer analysis of performance behavior of the system, the engine automatically captures per-request attributes such as end-to-end latency and existence of an error. By default, per-request attributes are written to a CSV-formatted file.

A.2.2 Plugin Usage

To use the plugin, developers implement a Go struct as part of the workflow specification that programmatically defines the workload. Since, Fig. A.3 shows the use of Blueprint's `workload` plugin for the ComposePost service from the social network application. The `ComplexWldGen` struct programmatically defines the workload. To define the workload, the `ComplexWldGen` struct implements the `RunComposePost` function (Lines 13-19) of type `RequestFunction` that executes requests to the ComposePost API by first generating the request arguments (Line 14) and then by executing the request and capturing its attributes (Lines 27-30,

13-22 for the helper method). The struct implements the `Runnable` interface so that it is launched correctly as part of the runtime namespace in the generated system. The `Run` function (Lines 34-40) implements the interface and as part of its execution uses the APIs provided by the `workload` plugin to define the workload (Lines 35-36) and then run the engine that executes the workload (Lines 37-39).