Mining microservice design patterns

Kamala Ramasubramanian*
Google
Sunnyvale, CA, USA
kamalaram@google.com

Eliana Phillips
University of California, Santa Cruz
Santa Cruz, CA, USA
esphilli@ucsc.edu

Peter Alvaro
University of California, Santa Cruz
Santa Cruz, CA, USA
palvaro@ucsc.edu

ABSTRACT

Building microservices based on design patterns is common practice. Due to the scale and dynamic nature of these applications, engineers usually only have an incomplete mental model of the system. We have developed a methodology that identify instances of well known patterns such as caching or fallbacks by analyzing traces of executions. This is in contrast with most prior work that analyzes source code to mine design patterns. Our preliminary results identifying instances of patterns of interest across several different applications is promising and we discuss the different directions we can explore in this space.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; • **Computer systems organization** → *Redundancy*.

ACM Reference Format:
Kamala Ramasubramanian, Eliana Phillips, and Peter Alvaro. 2022. Mining microservice design patterns. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3542929.3563472

1 INTRODUCTION

It is common practice to build microservices using well-understood design patterns [6, 11, 14, 15, 18]. Some examples include patterns for caching, fallbacks, and retries. A fallback, if configured, is invoked after an initial call to a service fails. Similarly, when the requested data is not found in the cache, a cache miss occurs and an additional call to the database is made. While patterns for fallbacks or caches are well-understood, their instantiation for a particular system is typically unknown.

Mining design patterns is an active area of research and has applications in program comprehension, feature identification, feature extraction, and assessing software quality. Instantiations of application-level patterns can also be used for debugging behavioral and performance issues. Recent work explores mining the architecture of microservice applications based on their Kubernetes deployments [16] to test if applications adhere to microservice design principles and refactor them as necessary.

While most prior work has focused on finding instances of such patterns by analyzing source code via static and dynamic analysis [1, 3, 8–10], there is a missing piece - finding patterns in source code does not guarantee that the system behaves as programmers intend them to. Additionally, engineers may not recognize instances of design patterns due to the scale and constantly evolving nature of systems, exacerbating the problem.

To gain confidence that systems behave as expected, we look for instances of design patterns in observations of executions. In our work, we consider patterns that arise from communication between participants in distributed executions. To find such instances of design patterns, we need to reason in aggregate across many executions. For example, only by observing pairs of executions can we establish that two services occur in mutually exclusive executions. Additionally, observing many executions allows us to trim false positives. If we postulate that service X and Y occur in mutually exclusive executions and subsequently observe a single execution containing calls to both service X and service Y, the instance is disqualified and not returned as a result that matches the pattern.

Our key insight is that querying observations of executions allows us to match templates of design patterns to their instantiations across different applications. We do so by factoring out application-specific details first following which we query the data for instances of design patterns. *Application experts* select and transform fields from traces into sets of tuples that are loaded into database tables based on a predefined schema. Then, *pattern experts* write SQL queries that are run against a database containing trace data

*Work was done pre-Google when author was a PhD student at University of California, Santa Cruz

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563472

from many executions. We describe our methodology in detail in the next section. We find instances of common design patterns - caching and fallbacks in sample microservice applications (HipsterShop [13], Deathstarbench [7], and applications from the Filibuster [12] corpus). Our preliminary results are promising and warrant further work in this space. In Section 2, we present our methodology with the fallback pattern as an example and discuss the system requirements that need to be satisfied for our techniques to apply. In Section 3, we discuss our results and in Section 4, we touch upon the most relevant related work. Section 5 highlights the applicability of our methodology to different problem domains and presents some examples.

2 METHODOLOGY

In our work, we focus on detecting two patterns in distributed execution traces: fallbacks and caching effects, templates of which are represented in Figure 1 (a). As can be seen, a fallback may be invoked in two main contexts. A fallback may be observed in a single execution when a call returns an error in response to which the caller then makes a call to a different service. In the template, potential fallbacks are identified by a failed call from service A to service B, followed by a successful call from service A to some other service, B', at a later time.

Alternatively, a fallback may be invoked when a call is dropped or lost in transit and triggers a timeout on the caller, which then makes a call to a different service. For this type of behavior, we need to look at *at least* two execution traces to confirm that this effect is indeed a fallback. In this template, the first successful execution contains a successful call, made from service A to service B. Contrast this with another successful execution, in which service A attempts to call service B but fails. Later, service A successfully calls service B'. Either call can occur, but not both, and it may be the case that neither of them occur.

We also are able to differentiate between cache hits and misses when observing executions. A cache hit occurs when a service attempts to retrieve data from cache and the data is present. A cache miss occurs when the data is not present in the cache, and the service must then retrieve data from the database at a later time. The basic template for cache miss is similar to fallback in single execution, but no call failures occur - instead, data is not found so more operations are needed to retrieve data.

While these patterns are simple, looking for them in practice is difficult, as distributed traces are often very large. Figure 1 (b) is a real (anonymized) trace from a production system. Since the templates of patterns we would like to find are small compared with individual traces, matching templates to instances of their occurrence manually is impractical. Next, we describe our methodology to automatically find instances of patterns.

Our methodology to discover patterns has two phases: an application specific phase to normalize trace data, and a query phase, which returns pattern instances. In the first phase, an application expert identifies the fields in the trace which need to be selected, transformed or discarded. Application experts always select fields such as service names, operation names, and error codes, but these may be named differently in various applications. Fields corresponding to service instance names, method names, file or line numbers may also be selected depending on the data recorded in the traces. A database containing normalized data corresponding to the set of executions is produced as output.

For example, all applications analyzed in this work utilize Jaeger tracing [2]. We have found that mining patterns in Jaeger traces requires selecting the service name, operation name, and status code labels, and transforming timestamp labels to logical time. Transformation of timestamp labels is encoded in the "index_of" function shown in Figure 2 (Map2) and is a mapping to the set of natural numbers which are monotonically increasing and represent logical time. Applying this mapping to all traces studied prepares them for further analysis in the subsequent phases. Application-specific expertise is required to write mappings for other tracing applications that uncovers these same patterns. We process the transformed trace data and load information about each event and call in each trace in the corpus into SQL tables. The result is a normalized data format which allows us to easily execute queries in the next phase.

In the second, query phase, a pattern expert writes queries in SQL to identify the patterns we are interested in from sets of traces such that the same query can be applied to processed trace data from different applications. For example, the fallback in a single execution can be identified by a failed call from service A to B followed by a later successful call from A to B'. A SQL query can identify this by selecting pairs of events with the following characteristics: exactly one failure and one success as sibling nodes, in which the failure occurs temporally before the success. A snippet of this query is shown on the extreme right in Figure 2.

To find instantiations of design patterns in traces of executions, we require that any pair of system executions is differentiated by at most one change, for a given set of executions. In our setting, a single change translates to failure of a call or crash of a service instance. This requirement is necessary since different changes can interact with each other leading to false positives that we cannot disambiguate. To satisfy this requirement, we are exploring a framework

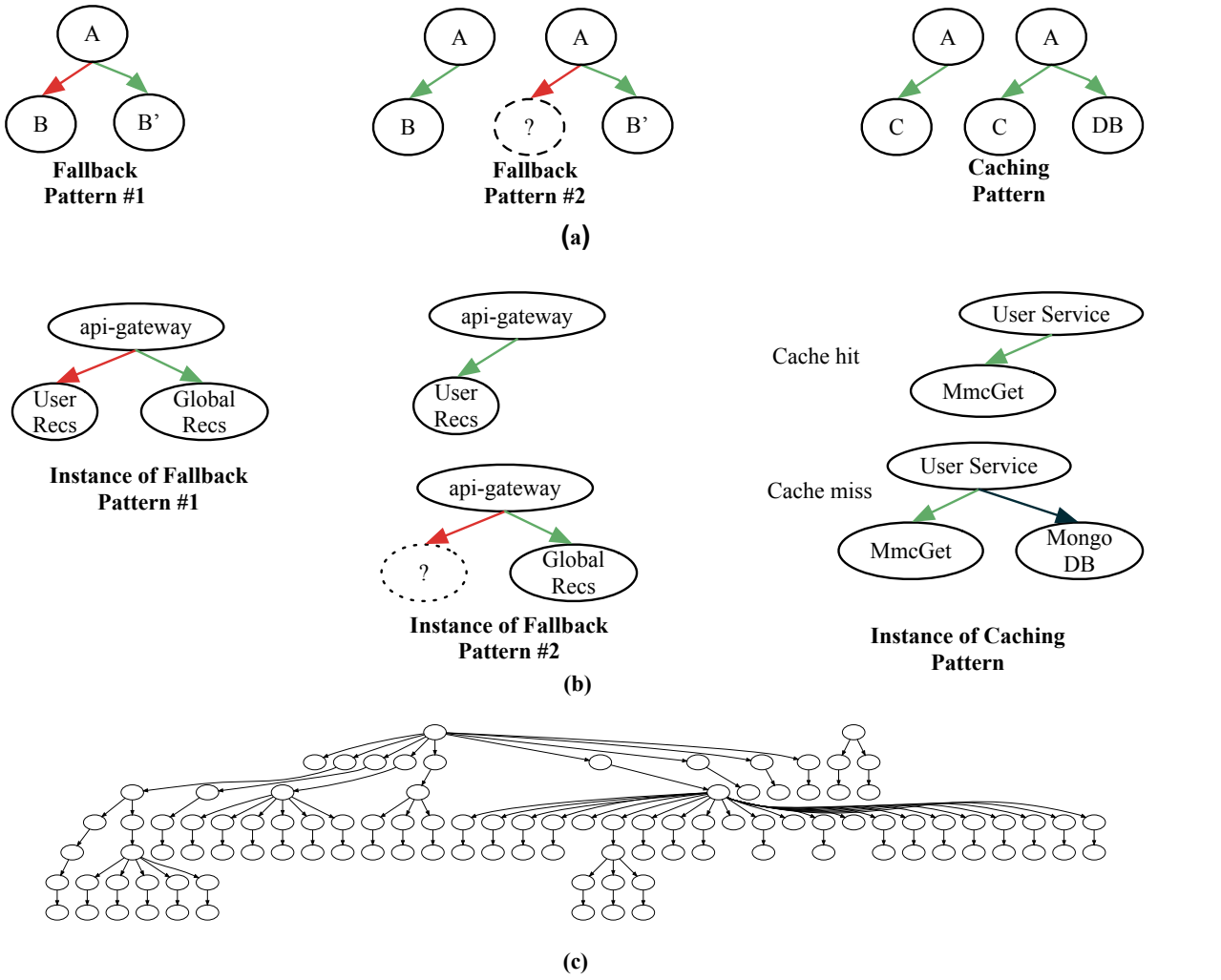

Figure 1: (a) represents common design patterns such as fallbacks and caching effects, where the red and green arrows represent failed and successful calls respectively. The dotted lines represent a service to which a call was attempted, but the message was dropped or lost in transit. (b) We demonstrate instances of fallback behavior in Netflix, as encoded by the Filibuster corpus and cache effects in Deathstarbench. (c) is an example trace taken from a real production system

that runs end-to-end tests repeatedly in a staging environment killing a process, injecting delays or mocking failures in different runs. Our framework also corrects for false positives as a result of non-deterministic effects of executions by witnessing traces of many executions and ordering results by decreasing frequency of their occurrence.

For our analysis, we consider sample microservice applications integrated with distributed traces - HipsterShop, Deathstarbench and applications in the Filibuster corpus. In the next section, we discuss our experimental methodology and preliminary results finding potential fallbacks and caching effects in different applications.

3 RESULTS

To find instances of fallbacks and caching templates in different applications, the set up consists of a few steps. First, we configure and run applications so that traces of executions are captured. Secondly, we identify functional tests to run or APIs to invoke that exercise desired functionality. We run the functional test or invoke the API at least once to capture traces during normal operation. Finally, we trigger fallback or caching behavior in applications via injecting crash faults or mocking errors in responses and run the functional test again, analyzing traces captured from executions

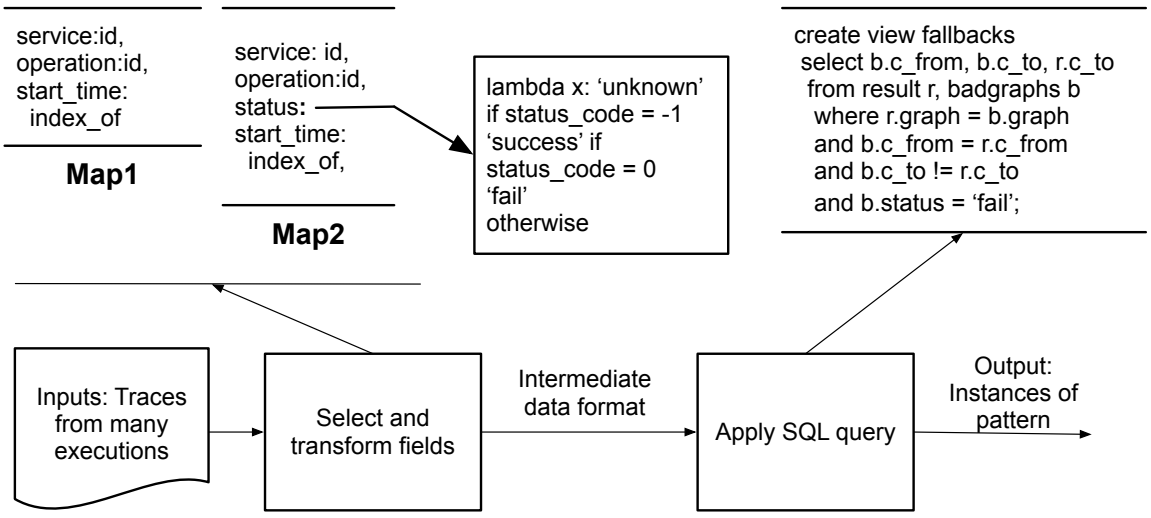

Figure 2: System workflow showing the steps in our methodology with a running example. The id in the mappings corresponds to identity, which means that the fields are retained as-is. index_of indicates that the start time is converted into a logical time and we have also shown how status code is mapped to one of three strings.

during normal system operation and when different faults are injected.

Table 1: Instances of patterns in different applications

	Fallback pattern # 1	Fallback pattern # 2	Caching
Hipstershop	✓	✓	
Cinema-6 (Filibuster)	✓	✓	
Netflix (Filibuster)	✓	✓	
Expedia (Filibuster)	✓	✓	
Deathstarbench			✓

In our setting, we have configured applications to send traces to Jaeger. As discussed in the previous section, we select the service name, operation name, and status code labels from each trace, and transform timestamp labels to logical time. Keeping these mappings fixed, we write different queries for each template we want to identify. As can be seen in Table 1, we found evidence of the two fallback patterns in several applications in the Filibuster corpus and were also able to confirm that the fallbacks we added programmatically to Hipstershop were discovered by our queries in executions. We also found evidence of caching effects in Deathstarbench executions captured by crashing instances of different caches and invoking specific APIs that reveal cache hits and misses.

4 RELATED WORK

Mining design patterns is an active area of research [3]. Most prior work analyzes source code to find patterns that have uses in program comprehension, feature identification, and feature extraction[citation]. While prior work attempts to find design patterns in source code by building and analyzing the abstract syntax tree (AST) corresponding to programs [8, 17], we attempt to find design patterns that demonstrate system function for microservice applications from execution structure. Many approaches use static and dynamic analysis [1, 3, 8–10] of source code and graph mining approaches, the latter of which are most relevant to our work.

Graph mining approaches [4, 5, 17] attempt to find isomorphic subgraphs to identify design patterns or anti-patterns in programs. Most of these approaches attempt to perform graph computations, which are computationally expensive, in contrast to our approach of formulating SQL queries with data from normalized traces as input. Additionally, these approaches attempt to find all recurring patterns, leaving interpretation to domain experts, while our approach is targeted to specific, well-understood patterns.

Recent work has also considered mining architectures based on deployments [16] to automatically infer architectures and find possible violations of microservice design patterns, as well as finding and fixing performance antipatterns via architectural refactoring. These approaches focus

their attention on finding and addressing violations in applications, in contrast to our approach of finding instantiations of design patterns in continuously evolving systems.

5 DISCUSSION

Our work in mining specific, common design patterns from distributed traces is unique, efficient, and covers a space of distributed design patterns research that is less explored in previous work. The system behaviors identified in this work can have a variety of applications, most notably in building domain knowledge, feature development, and debugging behavioral and performance issues. Some examples are:

(1) If we determine that some service X can serve as a fallback for both Y and Z, but a fallback has not yet been configured for Z, developers may configure X as a fallback for Z as well. Alternatively, if Z fails, traffic may be temporarily redirected to X to keep the system functional.
(2) If an increase in cache misses is observed at the same time as a performance regression, investigating the cache would be a good place to start.
(3) Finding examples of anti-patterns can help engineers proactively identify and fix issues before they cause a failure.

Retries follow a similar template to fallbacks within a single execution, except both calls are to the same destination, with the earlier call having failed. However, when querying for this pattern, we found that retries are observed when errors propagate up the trace graph when a fallback is invoked not by the immediate caller, but a service higher up. We speculate that retries that occur independently of fallbacks could represent an anti-pattern, especially if the retry is to the same service instance as the failed call.

Our first phase requires that application experts select and transform fields in traces to obtain a common set of labels that can be queried; a manual and tedious process. We posit that we can address this issue by automating the mapping process shown in Figure 2. These mappings are simple examples of trace abstraction, in which the size and complexity of traces are reduced by eliminating low-level details and preserving causal relationships and necessary information for trace comprehension. Successful usage of trace abstraction would allow our tool to tolerate variation in traces across applications and discrepancies within traces due to non-deterministic effects. A trace abstraction-like approach has been used in previous work in pattern-mining from traces to account for dynamic program behavior [9]. We seek to develop a unified approach to trace abstraction in future work.

6 CONCLUSION

Our techniques to identify instances of design patterns have found instances of fallbacks and caching effects in distributed traces in different applications. As discussed previously, finding these provides some evidence that the system functions as expected and has a variety of applications in building domain knowledge, feature development, and debugging. Future work in this space involves identifying and writing queries for more such patterns and anti-patterns as well as evaluating our techniques in a larger setting. Automatically finding mappings in the application-specific phase to reduce manual work for application experts is a challenging direction of future work as well.

Acknowledgements

This work was supported by the National Science foundation under Grant No. 1652368, and gifts from eBay and Meta.

REFERENCES

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring Hierarchical Motifs from Execution Traces. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery. https://doi.org/10.1145/3180155.3180216
[2] The Jaeger Authors. 2022. Jaeger Tracing. https://www.jaegertracing.io/
[3] B. Bafandeh Mayvan, A. Rasoolzadegan, and Z. Ghavidel Yazdi. 2017. The State of the Art on Design Patterns. *J. Syst. Softw.* 125, C (2017). https://doi.org/10.1016/j.jss.2016.11.030
[4] Hamid Abdul Basit and Stan Jarzabek. 2005. Detecting Higher-Level Similarity Patterns in Programs. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. Association for Computing Machinery. https://doi.org/10.1145/1081706.1081733
[5] Ahmed Belderrar, Segla Kpodjedo, Yann-Gael Gueheneuc, Giuliano Antoniol, and Philippe Galinier. 2011. Sub-Graph Mining: Identifying Micro-Architectures in Evolving Object-Oriented Software. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*. IEEE Computer Society. https://doi.org/10.1109/CSMR.2011.23
[6] Azure Architecture Center. 2022. Cloud design patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/
[7] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304013
[8] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. 2003. Automatic Design Pattern Detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC '03)*. IEEE Computer Society.
[9] Chun-Tung Li, Jiannong Cao, Chao Ma, Jiaxing Shen, and Ka Ho Wong. 2022. An agnostic and efficient approach to identifying features from

execution traces. *Knowledge-Based Systems* (2022).

[10] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2018. Detecting the Behavior of Design Patterns through Model Checking and Dynamic Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 4 (2018). https://doi.org/10.1145/3176643

[11] Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. 2018. A Pattern Language for Scalable Microservices-Based Systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings (ECSA '18)*. Association for Computing Machinery. https://doi.org/10.1145/3241403.3241429

[12] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. Association for Computing Machinery. https://doi.org/10.1145/3472883.3487005

[13] OpenCensus. 2019. Hipster Shop: Cloud-Native Microservices Demo Application. https://github.com/census-ecosystem/opencensus-microservices-demo

[14] Felipe Osses, Gastón Márquez, and Hernán Astudillo. 2018. Exploration of Academic and Industrial Evidence about Architectural Tactics and Patterns in Microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. Association for Computing Machinery. https://doi.org/10.1145/3183440.3194958

[15] Chris Richardson. 2019. *Microservice Patterns. With Examples in Java*. Manning, Shelter Island, NY.

[16] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. 2021. The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience* 51 (2021).

[17] Umut Tekin and Feza Buzluca. 2014. A Graph Mining Approach for Detecting Identical Design Structures in Object-Oriented Design Models. *Sci. Comput. Program.* 95, P4 (2014). https://doi.org/10.1016/j.scico.2013.09.015

[18] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. 2022. Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs. (2022). https://arxiv.org/abs/2201.03598