UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JHONNY MERTZ

Understanding and Automating Application-level Caching

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. PhD. Ingrid Nunes

Porto Alegre
March 2017

CIP — CATALOGING-IN-PUBLICATION

Mertz, Jhonny

Understanding and Automating Application-level Caching / Jhonny Mertz. – Porto Alegre: PPGC da UFRGS, 2017.

114 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Ingrid Nunes.

1. Web application. 2. Application-level caching. 3. Self-adaptive systems. 4. Software maintenance. 5. Software evolution. 6. Guideline. 7. Pattern. I. Nunes, Ingrid. II. Título.

"When something is important enough,

you do it even if the odds are not in your favor."

— ELON MUSK

ACKNOWLEDGEMENTS

My sincere gratitude to my family and friends, who always believed in me and supported me in my choices. To Simone, my girlfriend, thanks for the comprehension and affection, and for being my safe haven when I needed to reassure me.

I would like to thank my friends and colleagues from the Federal University of Rio Grande do Sul (UFRGS), for their help, support and friendship. A special thank you to my lab colleagues Fernando, Frederico, João, and Vanius, for their suggestions, ideas and our endless conversations.

I would also like to thank my advisor Ingrid Nunes for her guidance through the development of this dissertation and for serving as a reference of the researcher I want to be. Without her support and motivation this work would not be possible.

To those whose name I did not mention, but helped me developing this work in any way, thank you all. Finally, I must acknowledge that this research would not have been possible without the financial assistance of CNPq and the infrastructure provided by the Federal University of Rio Grande do Sul.

ABSTRACT

Latency and cost of Internet-based services are encouraging the use of application-level caching to continue satisfying users' demands, and improve the scalability and availability of origin servers. *Application-level caching*, in which developers manually control cached content, has been adopted when traditional forms of caching are insufficient to meet such requirements. Despite its popularity, this level of caching is typically addressed in an *ad-hoc* way, given that it depends on specific details of the application. Furthermore, it forces application developers to reason about a crosscutting concern, which is unrelated to the application business logic. As a result, application-level caching is a time-consuming and error-prone task, becoming a common source of bugs. This dissertation advances work on application-level caching by providing an understanding of its state-of-practice and automating the decision regarding cacheable content, thus providing developers with substantial support to design, implement and maintain application-level caching solutions. More specifically, we provide three key contributions: structured knowledge derived from a qualitative study, a survey of the state-of-the-art on static and adaptive caching approaches, and a technique and framework that automate the challenging task of identifying cache opportunities. The qualitative study, which involved the investigation of ten web applications (open-source and commercial) with different characteristics, allowed us to determine the state-of-practice of application-level caching, along with practical guidance to developers as patterns and guidelines to be followed. Based on such patterns and guidelines derived, we also propose an approach to automate the identification of cacheable methods, which is often manually done and is not supported by existing approaches to implement application-level caching. We implemented a caching framework that can be seamlessly integrated into web applications to automatically identify and cache opportunities at runtime, by monitoring system execution and adaptively managing caching decisions. We evaluated our approach empirically with three open-source web applications, and results indicate that we can identify adequate caching opportunities by improving application throughput up to 12.16%. Furthermore, our approach can prevent code tangling and raise the abstraction level of caching.

Keywords: Web application. application-level caching. self-adaptive systems. software maintenance. software evolution. guideline. pattern.

Entendendo e Automatizando Cache a Nível de Aplicação

RESUMO

O custo de serviços na Internet tem encorajado o uso de cache a nível de aplicação para suprir as demandas dos usuários e melhorar a escalabilidade e disponibilidade de aplicações. *Cache a nível de aplicação*, onde desenvolvedores manualmente controlam o conteúdo cacheado, tem sido adotada quando soluções tradicionais de cache não são capazes de atender aos requisitos de desempenho desejados. Apesar de sua crescente popularidade, este tipo de cache é tipicamente endereçado de maneira *ad-hoc*, uma vez que depende de detalhes específicos da aplicação para ser desenvolvida. Dessa forma, tal cache consiste em uma tarefa que requer tempo e esforço, além de ser altamente suscetível a erros. Esta dissertação avança o trabalho relacionado a cache a nível de aplicação provendo uma compreensão de seu estado de prática e automatizando a identificação de conteúdo cacheável, fornecendo assim suporte substancial aos desenvolvedores para o projeto, implementação e manutenção de soluções de caching. Mais especificamente, este trabalho apresenta três contribuições: a estruturação de conhecimento sobre caching derivado de um estudo qualitativo, um levantamento do estado da arte em abordagens de cache estáticas e adaptativas, e uma técnica que automatiza a difícil tarefa de identificar oportunidades de cache. O estudo qualitativo, que envolveu a investigação de dez aplicações web (código aberto e comercial) com características diferentes, permitiu-nos determinar o estado de prática de cache a nível de aplicação, juntamente com orientações práticas aos desenvolvedores na forma de padrões e diretrizes. Com base nesses padrões e diretrizes derivados, também propomos uma abordagem para automatizar a identificação de métodos cacheáveis, que é geralmente realizado manualmente por desenvolvedores. Tal abordagem foi implementada como um framework, que pode ser integrado em aplicações web para identificar automaticamente oportunidades de cache em tempo de execução, com base na monitoração da execução do sistema e gerenciamento adaptativo das decisões de cache. Nós avaliamos a abordagem empiricamente com três aplicações web de código aberto, e os resultados indicam que a abordagem é capaz de identificar oportunidades de cache adequadas, melhorando o desempenho das aplicações em até 12,16%.

Palavras-chave: Aplicações web. cache. sistemas adaptativos. manutenção de software. evolução de software. diretrizes. padrões.

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Autonomic Computing
BHR	Byte Hit Ratio
CDN	Content Delivery Network
DNS	Domain Name Server
GDS	Greedy-dual Size
HR	Hit Ratio
IP	Internet Protocol
ISP	Internet Service Provider
LFU	Least Frequently Used
LRU	Least Recently Used
LSR	Latency Saving Ratio
ML	Machine Learning
TRP	Throughput
TTL	Time-to-live

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Dynamically generated web content represents a significant portion of web requests, and the rate at which dynamic documents are delivered is often orders of magnitudes slower than static documents (RAVI; YU; SHI, 2009). To continue satisfying users' demands and also to reduce the workload on content providers, over the past years many optimization techniques have been proposed. The ubiquitous solution to this problem is some form of *caching* (PODLIPNIG; BÖSZÖRMENYI, 2003). Recently, latency and cost of Internet-based services are driving the proliferation of *application-level caching*, in which developers manually insert caching logic into the application base code to decrease the response time of web requests by temporary saving frequently requested or expensive to compute content in memory. Therefore, this type of caching has become a popular technique for enabling highly scalable web applications.

Although its popularity, the implementation of application-level caching is not trivial and demands high effort, given that it is typically implemented in an *ad hoc* way. It involves four key challenging issues: determining *what* data should be cached, *when* the data selected should be cached or evicted, *where* the cached data should be placed, and *how* to implement the cache efficiently. The main problem is that solutions for all of these issues usually depend on application-specific details and are manually designed and implemented by developers, which can be time-consuming, error-prone and, consequently, a common source of bugs (PORTS et al., 2010; GUPTA; ZELDOVICH; MADDEN, 2011).

Furthermore, developers must continuously inspect application performance and revise caching design choices, due to changing workload characteristics and access patterns. Consequently, initial caching choices may become obsolete (CHEN et al., 2016). However, such maintenance is compromised due to the caching logic being tangled with the business logic, which implies even more extra time dedicated to maintenance (RADHAKRISHNAN, 2004).

Therefore, all these issues call for new approaches to support developers in developing application-level caching, and also provide an optimal utilization of the web infrastructure, in particular of the caching system. Such approaches can provide a better experience with caching for developers. In the remaining sections, we present the research problem addressed by this dissertation and the limitations of existing work in Section 1.1. Then, we describe our proposed solution and present an overview of the contributions of this work in Section 1.2. Finally, Section 1.3 details the structure of the remainder of this dissertation.

1.1 Problem Statement and Limitations of Related Work

Given the issues involved with application-level caching, our research question consists of *how to support developers while designing, implementing and maintaining application-level caching in web application and reduce the effort they spend on these activities?*

Although application-level caching is commonly being adopted, it is typically implemented in an *ad hoc* way, and there are no available practical guidelines for developers to appropriately design, implement and maintain caching in their applications. To find caching best practices, developers can make use of conventional wisdom, consult development blogs, or simply search online for tips. Nevertheless, this empirical knowledge is unsupported by concrete data or theoretical foundations that demonstrate its effectiveness in practice. Although a well designed and implemented cache can achieve desired non-functional requirements, application performance may decay over time due to changes in the application domain, workload characteristics and access patterns. Therefore, achieving acceptable application performance requires constantly tuning cache decisions (RADHAKRISHNAN, 2004).

Existing research on application-level caching focuses on automating cache-related tasks or at least guiding developers while developing an application-level caching solution. Thereby, despite substantial advances have been made towards supporting developers in this challenging task, there still exist limitations that remain unaddressed. Such limitations are discussed next.

Complex logic and personalized web content remain unaddressed. Required cache-related tasks, such as the identification of caching opportunities, have received support in the context of web pages (NEGRÃO et al., 2015), and database queries and objects (CHEN et al., 2016). However, complex logic and personalized content, which are produced and handled by the application, remain unaddressed.

Most of the existing approaches do not consider application-specific details. Even though there are proposed solutions to application-level caching, application specificities are often ignored by these solutions while addressing caching issues. Such solutions usually optimize cache decisions based on cache access information (size, recency and frequency) and cache statistics (hit and miss ratios), thus ignoring cache meta-data expressed by application-specific characteristics, which are closely related to application computations and could help to reach an optimal performance of the caching service on time. When they consider application specificities, a common approach to acquire such information is to require from developers the provision of knowledge associated with the semantics of application code and data. In real world applications, providing this information becomes an expensive task, which is also affected by the following limitation.

There is a lack of seamlessness and automation in the popular existing tools. Although there are existing tool-supported approaches that can help developers implement caching with minimal impact on the application code (PORTS et al., 2010; ZHANG; LUO; ZHANG, 2015), they only consider implementation issues, letting most part of the reasoning, as well as the integration with the tool, to developers.

There is a lack of adaptation in proposed cached solutions. Most of the existing work does not provide adaptive approaches to deal with caching issues, which turns such approaches sensitive to the application changing dynamics. Furthermore, recent studies have shown that approaches that apply some reasoning to cache-related decisions are more efficient when compared to static approaches (ALI; SHAMSUDDIN; ISMAIL, 2011). This shortcoming motivates the proposal of adaptive caching solutions, and although there are many studies involving web caching, the enhancement of such solutions by using adaptive techniques is still barely explored.

Thus, such limitations call for new methods and techniques, which can support developers to reach a good trade-off between cache development effort and application performance improvement.

1.2 Proposed Solution and Overview of Contributions

An essential step towards answering our research question to address the challenges imposed on developers is to *understand*, *extract*, *structure* and *document* the application-level caching knowledge implicit and spread in existing web applications. In order to do so, we performed a qualitative study, which involved the investigation of ten (open-source and commercial) web applications with different characteristics, to identify patterns and guidelines to support developers while designing, implementing and maintaining application-level caching. The results of this study are thus informative and provide practical guidance for developers, given that there is neither off-the-shelf solutions or systematic ways of designing, implementing, and maintaining it.

Although the proposed guidelines and patterns can help developers while handling cache into their applications, even a well designed and implemented cache, which achieves the desired non-functional requirements, may decay in terms of application performance over time due to changes in the application domain, workload characteristics and access patterns. We thus propose an automated framework that manages the cache according to observations made by monitoring a web application at runtime, as well as detaches caching concerns from the application. As opposed to traditional caching approaches, which cache content such as web pages or database queries, our approach focuses on caching *method-level content*. Moreover, we use *application-specific information* to make caching decisions, such as the user session.

Our proposed solution can be incorporated into web applications, so that it can monitor and choose content to be cached according to changing usage patterns. Because it monitors the application at runtime, it is sensitive to the evolution of application workload and access patterns, thus being able to self-optimize caching decisions. Alternatively, it serves as a decision-support tool to help developers in the process of deciding what to cache, guiding them while manually implementing caching. We evaluated our approach empirically with three open-source web applications, which have different domains and sizes. Obtained results indicate that our approach can identify adequate caching opportunities by improving application throughput up to 12.16%.

In summary, we provide the following contributions:

- a survey that conveys the current state-of-the-art research on static and adaptive application-level caching approaches;
- a qualitative study that provides an in-depth analysis of how developers employ application-level caching in web-based applications;
- guidelines and patterns for caching design, implementation and maintenance to be adopted under different circumstances while modeling an application-level caching component;
- a caching approach focused on integrating caching into web applications in a seamless and straightforward way, providing an automated and adaptive management of cacheable methods; and
- a framework that detaches caching concerns from application code.

1.3 Outline

The remainder of this dissertation is organized as follows. In Chapter 2 we give an introductory background to application-level caching, detailing the issues involved with this type of caching. We detail the performed qualitative study and discuss its results in Chapter 3, while derived guidelines and patterns are presented in Chapter 4. We detail the proposed approach in Chapter 5, then present and discuss its evaluation in Chapter 6. Finally, we discuss related work, which provides different techniques to the identification and caching of cacheable opportunities, and we conclude and detail future work in Chapter 8.

2 BACKGROUND

In this section, we provide foundations on web caching and introduce principles of self-adaptive systems, presenting definitions and explanations of associated terms and concepts. This gives the required knowledge for understanding the work presented in later chapters.

2.1 Web Caching

To introduce web caching, we use the taxonomy presented in Figure 2.1, which groups web caching research into three main categories, each split into subcategories. The first category, *location*, is associated with where caching is located in the typically used web architecture. The second is concerned with how the caching is *coordinated*, which involves different tasks. Finally, the last is associated with how caching can be *measured*, to have its effectiveness assessed. We next explain each of these categories and their subcategories. Every caching solution can be classified according to these three categories, given that its development involves choosing a location, dealing with coordination issues, and selecting appropriate metrics to estimate the performance of the provided solution.

2.1.1 Location

Given that web caching is closely related to the web infrastructure components, we overview such architecture in Figure 2.2 along with the main caching locations, which are explored later. It presents a typical web architecture, which is widely adopted in practice and can roughly be separated into three components: client, Internet, and server (LABRINIDIS, 2009; RAVI; YU; SHI, 2009).

The *client* component is essentially the end user's computer and web browser; while the *Internet* component contains a plethora of different, interconnected mechanisms to enable the connection between the client and the server. A *Domain Name Server* is typically used to decode the name of the web address into an Internet Protocol (IP) address. With this IP address, routers are used so that the client can establish a connection to the server, using the HTTP protocol. The IP address of a web server can be differentiated according to the end user's IP address to take advantage of server proximity, when there

Figure 2.1: Web Caching Taxonomy.

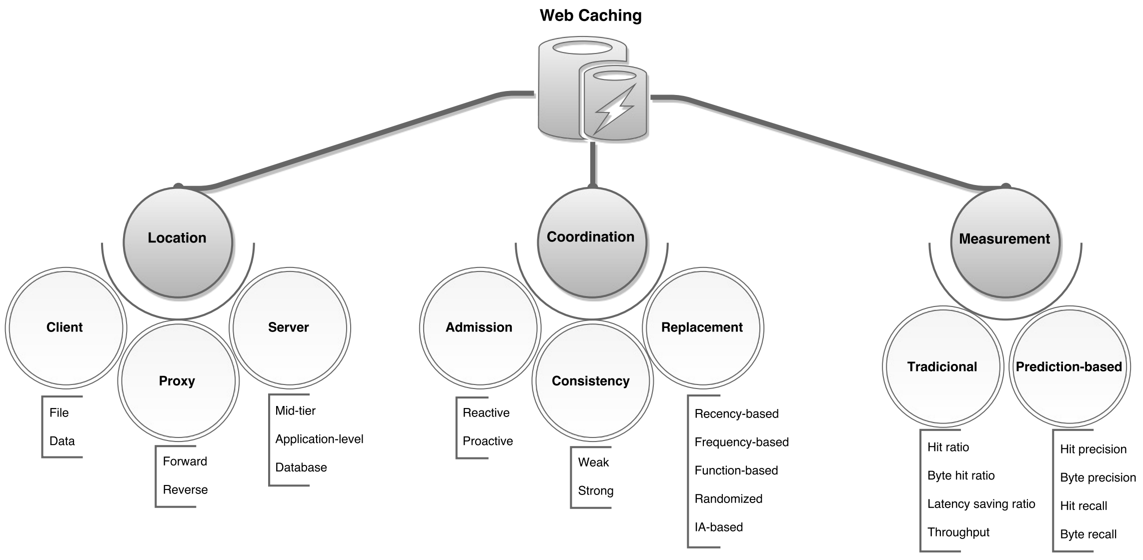
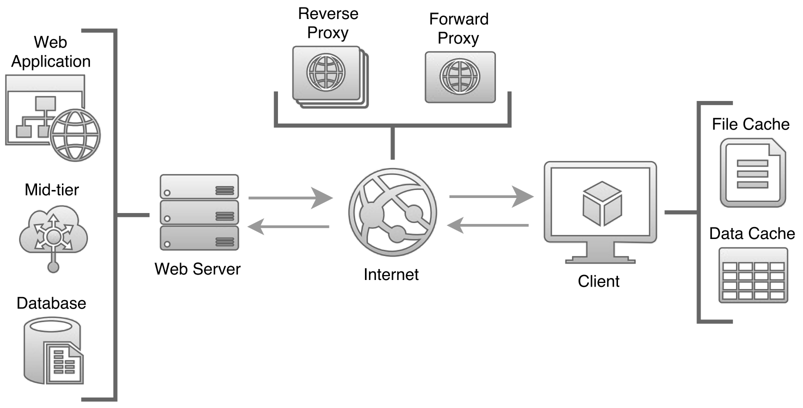


Figure 2.2: Traditional Web Application Architecture with Associated Caching Locations.



are alternatives due to replication. The *server* essentially includes multiple and different servers that are collectively seen as the web server by the end-user. In particular, the entry point for a cluster of servers would normally be a load balancer or web switch that is routing incoming web traffic to the different web servers available in the cluster. The entry point for a single web server is the web server itself. The web server is responsible for handling and responding to HTTP requests, and typically hosts a web application, which provides the business services. The web application is usually connected to back-end enterprise information systems (i.e. databases, file systems or other applications) and performs read and write operations driven by business rules (LABRINIDIS, 2009; RAVI; YU; SHI, 2009). In summary, caching of web content can be deployed at several locations across the typical web architecture, shown in Figure 2.2, ranging from the database to the client's browser (PODLIPNIG; BÖSZÖRMENYI, 2003). We classify web caching locations into three main locations, as follows.

Server Server-side caching solutions include all available caching locations within the server component, such as in databases, between database and application, within the application, and in the web server (RAVI; YU; SHI, 2009).

Proxy Located between client and server, a proxy caching can be employed in a forward (proxy on behalf of clients) or reverse (proxy on behalf of servers) way (PODLIPNIG; BÖSZÖRMENYI, 2003; RAVI; YU; SHI, 2009).

Client At the client level, browser data, and client-side computations can be cached near the client, more specifically, on a machine where the users' web browser is located (WANG, 1999; PODLIPNIG; BÖSZÖRMENYI, 2003; BALAMASH; KRUNZ, 2004).

Usually, caching solutions are designed and implemented as an abstraction layer that is transparent from the perspective of neighboring layers. Database and proxy caching are well-known for following this design principle. Differently, application-specific caches at the server-side or client-side require an extra effort from developers to be integrated into the web infrastructure (GUPTA; ZELDOVICH; MADDEN, 2011; WANG et al., 2014; PORTS et al., 2010).

Each caching location has its own benefits, challenges, and issues, leading to trade-offs to be resolved when choosing a caching solution. For instance, according to a selected choice of location, particular forms of data can be stored, such as the final HTML page (CANDAN et al., 2001), intermediate HTML or XML fragments (RAMASWAMY et al., 2005; LI et al., 2006; GUERRERO; JUIZ; PUIGJANER, 2011), application objects (PORTS et al., 2010; GUPTA; ZELDOVICH; MADDEN, 2011), database queries (SOUNDARARAJAN; AMZA, 2005; AMZA; SOUNDARARAJAN; CECCHET, 2005; BAEZA-YATE et al., 2007; MA et al., 2014), or database tables (LARSON; GOLDSTEIN; ZHOU, 2004). Furthermore, support varies across different locations, as well as hit and miss probabilities. Therefore, it is important to make an effort to achieve the best design rationale to optimize each caching solution according to different circumstances. Given these several available caching locations and their pros and cons, we summarize this information in Table 2.1.

Due to the variety of application domains, workload characteristics and access patterns, there is no universal best caching solution, because it is hard, if not impossible, to achieve a deployment scheme that performs best in all environments and maintains its

Table 2.1: Classification of Popular Web Caching Techniques based on the Location of Cache.

	Type	**Content Granularity**	**Advantages**	**Disadvantages**
Client-side caching	File	Static resources, HTML pages or page fragments	–Integrated into the user browser –Does not require developer efforts –Effective for static content	–Cached content becomes stale and cannot be shared –Requires manual response configurations in the server –Lower hit-ratios, but a hit provides a large gain
	Data	Dynamic requests from a single client to many servers	–Caches many servers for a single user –Reduces latency perceived by users	–Manually implemented –Cached items cannot be shared
Proxy-based caching	Forward	Content requests from many clients to many servers in the gateway server	–Many users share resources from the nearest proxy servers or their neighboring caches –Leads to wide area bandwidth savings, improved latency –Increases the availability of static content	–Requires a large-sized memory –May serve stale content –Difficult to maintain consistency
	Reverse	Content requests from many clients to one server	–Reduces bandwidth requirements –Reduces delay of page generation –Reduces the web server workload –Transparent to the application –Easy to set up, web servers should only support few configurations	–May serve stale content –Limited reusability –Requires a large-sized memory –Lower hit ratio
Server-side caching	Database	Tables, queries or views that are managed by the same database management system (DBMS) instance as the original database	–Addresses delays associated with databases –Improves performance, scalability, manageability of database drivers –Cached data are served to multiple web applications –Higher hit-ratios –Easy to maintain consistency	– Does not address other delays in dynamic web page generation and distribution
	Mid-tier	Partial or full database queries, service requests or any content processed between source of information and application	–Achieves greater reusability –Automatically caches and maintains consistency –Reduces the load on the database server –Reduces the load on database or services that are difficult to scale up	– Offers no direct benefit for customized load within the application
	Application-level	Entire HTML pages, page fragments, database queries or even computed results	– Arbitrary content –Can guarantee freshness –Saves processing request –Flexible implementation –Takes application specificities into account –Reduces application workload	– Requires specific content caching approach –Manually implemented

performance over an extended period. Intuitively, caching at the database stage typically offers higher hit ratios, while caching at the web page generation stage offers greater benefits in the case of a hit. Therefore, caching at different locations is complimentary, and a combination of different solutions is possible, and can bring even more benefits with the cost of a higher effort invested in caching configuration (AMZA; SOUNDARARAJAN; CECCHET, 2005; RAVI; YU; SHI, 2009; SIVASUBRAMANIAN et al., 2007).

Regardless of the cache location, coordination techniques should be employed to allow caching to improve the web-based system performance. Coordination topics are explored next.

2.1.2 Coordination

As discussed above, caches may be placed in several locations and, regardless of where and how the cache is implemented, issues such as the limited cache space should be addressed to ensure the cache efficiency. Cache coordination addresses these issues. It includes management strategies, which are designed to decide adequate content to be cached, the right moment of caching or clearing the cache to avoid stale content, and to deal with situations where the cache gets full of content and there is a new item to be added. Such strategies can be considered in any caching location. Coordination issues and associated strategies can be split into three main groups, namely admission, consistency, and replacement, which are detailed next.

Admission Admission policies are used for content selection and preventing caching of unimportant content (BAEZA-YATE et al., 2007; EINZIGER; FRIEDMAN, 2014; SULAIMAN et al., 2009). They can have two key goals: (a) prevent unnecessary items from being inserted into the cache, and (b) predict web objects expected to be requested in the (near) future and insert them into the cache. The former focuses on employing a *reactive* mechanism to filter content and admitting only valuable web objects in the cache. The latter proactively prefetches and caches content expected to be requested, which can be seen as a *proactive* admission policy, improving significantly cache performance and reducing the user perceived latency (GAWADE; GUPTA, 2012; ALI; SHAMSUDDIN; ISMAIL, 2011; DOMÈNECH et al., 2006). Prefetching usually exploits the spatial locality shown by web objects (e.g. correlated reference in different documents) and takes advantage of component's (server, proxy or client) idle time, preventing bandwidth underutilization and hiding part of the latency (GAWADE; GUPTA, 2012; Veena Singh Bhadauriya, Bhupesh Gour, 2013). Ali, Shamsuddin and Ismail (2011) provided a comparison of the server, client, and proxy prefetching challenges and issues. They also surveyed prefetching studies at the proxy level, and classified these studies into two broad categories (content-based and history-based), according to the data used for prediction.

Consistency Approaches that address consistency deal with the fact that every piece of cached data is already potentially stale, because cached items can be updated at their source. Therefore, if there are freshness requirements, it is necessary to design and implement consistency approaches to avoid staleness of cached items. There are two main consistency models to caching: (a) *weak*, which provides higher availability by relaxing consistency (i.e. stale data is allowed to be returned from the cache); and (b) *strong*, which ensures that stale data is never returned by a higher cost of processing. Both models can be achieved through validation or invalidation processes. With validation, the application explicitly invalidates pages when they are modified by verifying the cached items in their source, e.g. with the origin server. With invalidation, the system can track data dependencies to identify which items need to be invalidated when the underlying database is modified, or the source server notifies the cache system if a cached item has changed. Furthermore, if weak consistency is adopted, timeouts—i.e. time to live (TTL)—can be used to expire cached items periodically. Traditionally, cache consistency (or coherence) is treated as a separate issue from web caching, being widely studied, including in the area of distributed systems (GHANDEHARIZADEH; YAP; BARAHMAND, 2012; PORTS et al., 2010; AMZA; SOUNDARARAJAN; CECCHET, 2005; SIVASUBRAMANIAN et al., 2007; GAO et al., 2005).

Replacement In situations when the cache achieves its full capacity, and there is a new item to be added to the cache, replacement policies are responsible for deciding which one should be removed. Such policies ideally remove items that are less likely to be a hit, i.e., to be requested while they are in the cache. Traditional replacement strategies are classically classified into four categories: recency-based, frequency-based, function-based and randomized (PODLIPNIG; BÖSZÖRMENYI, 2003). Recently, the availability of monitoring workload and user accesses, and the dissemination of mechanisms to store and process such information have motivated the proposal of intelligent replacement policies, which exploit such data as training data to build a meta-model (ALI; SHAMSUDDIN; ISMAIL, 2011). Despite that the most popular policy is the *least recently used*, due to its simplicity and relatively good performance, several other replacement approaches have been proposed with the aim of getting good performance in many situations (WONG, 2006), and almost all of them were demonstrated to be superior to others in their proposal (PODLIPNIG; BÖSZÖRMENYI, 2003; Ali Ahmed; SHAMSUDDIN, 2011; SULAIMAN; SHAMSUDDIN; ABRAHAM, 2013). These results indicate that there is no best policy in all scenarios, because a policy performance depends on workload characteristics. Focusing on these differences, Wong (2006) provides a pragmatic approach to choosing the best policy. In summary, each policy imposes a trade-off between success rate and computational overhead.

Many factors can be observed while choosing or proposing a coordination strategy. Coordination strategies usually evaluate temporal locality properties, such as recency (i.e. time of the last reference to the item) and frequency (i.e. the number of previous requests to the item) (PODLIPNIG; BÖSZÖRMENYI, 2003). In addition to temporal locality, the spatial locality can also be explored by analyzing the content context, which is not always straightforward. Other factors include size, access latency and time since last modification of the content, or even heuristics, such as useful time of a cached item. However, taking into account all factors to achieve an improved coordination decision is not a trivial task, given the computational overhead. Furthermore, there are no well-accepted influence factors, because all of them are subject to the particularities of the environment requirements, which means that the importance of properties depends on the scenario or environment under consideration (PODLIPNIG; BÖSZÖRMENYI, 2003; WONG, 2006).

In summary, all coordination strategies aim to solve caching issues to improve the general performance of web-based applications. However, cache benefits come with the cost of investing an effort to design, implement, and maintain the cache. The benefits of each caching strategy can be estimated regarding improvements to the performance of the cache and the system, which is measured with different metrics that are described next.

2.1.3 Measurement

The last category of our taxonomy for web caching is related to measurement techniques, which are used to measure the efficiency of web caching strategies (PODLIPNIG; BÖSZÖRMENYI, 2003; ALI; SHAMSUDDIN; ISMAIL, 2011). The most popular and well-accepted metrics are summarized in Table 2.2 with their name and acronym, description and definition of the metric. Similarly to coordination strategies presented in the previous section, cache performance metrics can be applied in any caching scheme, being even not limited to web caching. Furthermore, these metrics are well-known and generic enough to evaluate and compare any caching strategy and tuning task.

Note that HR and BHR conflict with each other, given that keeping small popular items in the cache optimizes HR, whereas holding large popular items optimizes BHR. Moreover, because it is hard to measure the download time of items precisely and the response time (it is affected by many factors independent from the cache, such as network congestion and server stability), LSR and TRP require a well specified performance test

Table 2.2: Summary of Traditional Cache Evaluation Metrics.

Metric (Acronym)	**Description**	**Definition**
Hit Ratio (HR)	HR is the percentage of requests that can be satisfied by the cache. The higher the hit ratio, the better the technique employed.	$\sum_{i \epsilon R} \frac{h_i}{f_i}$
Byte Hit Ratio (BHR)	BHR is concerned with how many bytes are saved, i.e. the number of bytes satisfied from the cache as a fraction of the total bytes required by clients.	$\sum_{i \epsilon R} s_i \cdot \frac{h_i}{f_i}$
Latency Savings Ratio (LSR)	LSR is defined as the ratio of the sum of request time of item satisfied by the cache over the sum of all requesting time.	$\sum_{i \epsilon R} d_i \cdot \frac{h_i}{f_i}$
Throughput (TRP)	The throughput is measured by calculating the number of requests per second throughout a period of time. A higher throughput shows the effectiveness of the caching, as more requests can be satisfied within the same period of time.	$\sum_{i \epsilon R} \frac{f_i}{t_i}$
Notation: s_i = size of item i f_i = total number of requests for item i h_i = total number of hits for item i t_i = total fetch delay in seconds of requests for item i d_i = mean fetch delay from server for item i R = set of all requests		

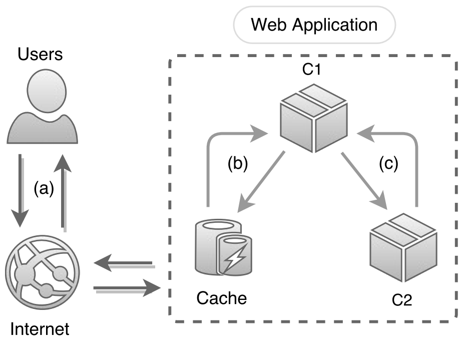
and due to this are not widely used (WONG, 2006).

2.2 Application-level Caching

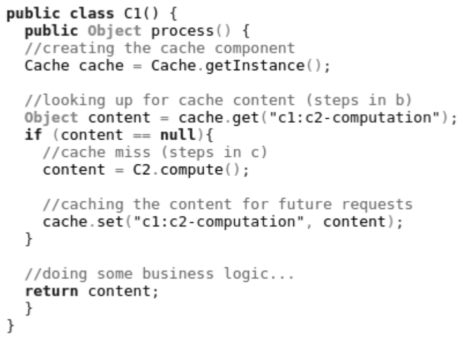
As opposed to caching alternatives that are placed outside of the application boundaries, application-level caching allows storing content at a granularity that is possibly best suited to the application. For example, many websites today provide highly-personalized content, thus rendering whole-page web caches is largely useless. Therefore, application-level caching can be used to separate common from customized content at a fine-grained level, and then the common content can be cached and shared among users (PORTS et al., 2010).

In order to understand the issues and the approach presented in this work, we provide an introductory example in which application-level caching is used to lower the application workload in Figure 2.3. It details the process of using the cache and how it is implemented. First, a web application receives an HTTP request from a user (*step a*), which is eventually treated by a component C1. However, C1 depends on C2, and calling and executing C2 may imply an overhead regarding computation or bandwidth. Therefore, C1 manages to cache C2 results and, for every request, C1 verifies whether C2 should be called or there are previously computed results already in the cache (*step b*). Then, the cache component, which is typically a key-value in-memory storage, performs

Figure 2.3: Application-level Caching Overview.



(a) Process Steps.



```
public class C1() {
  public Object process() {
  //creating the cache component
  Cache cache = Cache.getInstance();

  //looking up for cache content (steps in b)
  Object content = cache.get("c1:c2-computation");
  if (content == null){
    //cache miss (steps in c)
    content = C2.compute();

    //caching the content for future requests
    cache.set("c1:c2-computation", content);
  }

  //doing some business logic...
  return content;
  }
}
```

(b) Code Example.

a look up for the requested data and returns either the cached result or a not found error. If a *hit* occurs, it means the content is cached, and C1 can avoid calling C2. However, when a *miss* occurs and a not found error is returned, it means that C2 computation is required (*step c*). The newly fetched result of C2 can then be cached to serve future requests faster.

The steps described in the Figure 2.3(a) are those typically performed at any cache implementation. Furthermore, other caching layers can be deployed outside the application such as proxy or database caches. The key difference is that, in application-level caching, the responsibility of managing the caching logic is entirely left to application developers, possibly with the support of frameworks that provide an implementation of the cache system, such as EhCache[1] and Memcached[2].

2.2.1 Challenges and Issues

As stated in the introduction, the development of application-level caching involves four key issues: determining *how*, *what*, *when*, and *where* to cache data. The first issue is related to the usual *cache-aside* implementation, and the fact that the cache system and the underlying source of data should be connected and handled by the application, as shown in Figure 2.3(a). Therefore, developers must manually implement ways to *assign keys* to cached items, perform lookups, and keep consistency between the cache and the application. Such logic is placed within the application and is usually tangled with the business logic—see Figure 2.3(b), in which decisions are explicit (i.e. which objects to get, put or remove from the cache), as opposed to an implicit cache, in which the cache is implemented as a transparent layer.

[1]<http://www.ehcache.org/>
[2]<https://memcached.org/>

Caching implementation and maintenance are thus a challenge, because caching becomes a *cross-cutting concern*, spread all over the code—mixed with business code—which causes increased complexity and maintenance time (PORTS et al., 2010). Furthermore, the extra caching logic also requires *additional testing* and *debugging time*, which can be expensive in some scenarios (MERTZ; NUNES, 2016a). Moreover, web applications are usually not conceived to use caching since the beginning. As the application evolves, complexity or usage analysis are performed and may lead to requests for improvements (RADHAKRISHNAN, 2004). Thus, developers must *refactor* data access logic to encapsulate cache data into the proper application-level object. Although it is often not possible to predict that an application will require application-level caching in the future, posterior cache implementation can lead to significant rework due to refactoring and an implementation that would have better quality if thought before the application reaches advanced stages of development.

The second and third issues are associated with the decision of the right moment to cache or clearing the cache to avoid cache thrashing and stale content. These decisions involve (a) the choice for the granularity of cache objects; (b) translation between raw data and cache objects; and (c) maintenance of the cache consistency, which are all tasks to be accomplished by developers and might not be trivial in complex applications. Therefore, it is crucial to understand what are the typical usage scenarios, how often the data selected to be cached is going to be requested, how much memory this data consumes, and how often it is going to be updated. Furthermore, any practical cache design reflects a compromise between implementation cost and application performance improvements. Finally, the last issue is related to the management of the cache system, which involves several other *non-trivial decisions*, such as determining replacement policies and the size of the cache.

A fundamental problem of application-level caching is that solutions for all issues aforementioned usually depend on application-specific details and, consequently, solutions are manually designed and implemented by developers. This on-demand and manual caching process can be very time-consuming, error-prone and a common source of bugs (PORTS et al., 2010; GUPTA; ZELDOVICH; MADDEN, 2011).

2.2.2 Static vs. Adaptive Application-level Caching

In order to provide solutions to deal with caching that require less intervention, thus easier and faster to be adopted, static approaches have been proposed to help developers while designing, implementing and maintaining an application-level caching solution. These mainly address implementation issues, such as raising the abstraction level of caching and automating some of the required tasks. *Static approaches* usually focus on decoupling this non-functional requirement from the base code and providing ready-to-use caching components with basic functionalities and default configurations, such as a storage mechanism and standard replacement policies. However, even when leveraging static solutions to ease the development of a caching system, the challenges and issues related to the design and maintenance of this caching system remain unaddressed, because default configurations do not cover all the design issues, and those provided may even not perform well in all contexts.

Therefore, developers must still specify and tune cache configurations and strategies, taking into account application specificities. According to Radhakrishnan (2004), a traditional approach to design and maintain cache solutions is to set up strategies based on assumptions or well-accepted characteristics of workload and access patterns. However, given that these strategies are not based on application specificities, cache and application performance may decay over time due to changes in the application domain, workload and user accesses. Then, cache statistics, such as hit and miss ratios, can be observed and used to decide whether to change first choices. This process is repeated until an acceptable trade-off between configuration and performance is reached, and then cache management strategies are fixed for the lifetime of the product or at least for several release cycles. As a result, to achieve caching benefits so that the application performance is improved, it is necessary to constantly tune cache decisions, which implies even more extra time dedicated to maintenance. Despite the effort to tune caching decisions, it is not possible to keep maintenance tasks up for a long time and, eventually, an unpredicted or unobserved usage scenario may emerge. As the cache is not tuned for these situations, it would likely perform sub-optimally.

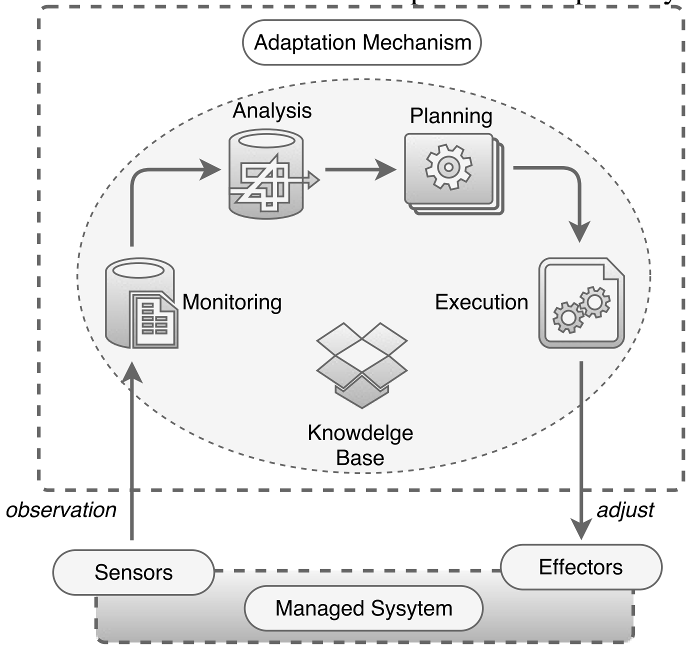
This shortcoming motivates the need for *adaptive caching solutions*, which could overcome these problems by adaptively adjusting caching decisions at run-time to maintain a required performance level. Moreover, an adaptive caching solution would minimize the challenges it poses for developers, requiring less effort and providing a better experience with caching for them, as opposed to providing a simple key-value store that must be manually managed and tuned to address assumed application bottlenecks. In addition, such adaptive caching solutions aim at optimizing the utilization of the infrastructure, in particular for the caching system. Next, we present basic concepts regarding adaptive systems that are used in later chapters of this work.

2.3 Adaptive and Self-adaptive Systems

Adaptive systems in general respond to changes in their internal state or external environment with the guidance of an underlying control system. Particularly, caching systems are likely to require dynamic adaptation because of the stochastic nature of user behavior. Therefore, since their conception, it was desirable to make caching solutions conform to the dynamic changing of user demands and the network environment. Traditional cache strategies, such as replacement algorithms, already present a mechanism for adapting themselves over time, based on general heuristics like recency or frequency (WANG, 1999). However, the term *adaptive caching* is slightly overloaded. Although these solutions can somehow be seen as adaptive, because they exhibit some form of feedback, they are conceived and implemented *statically*. For instance, most of these solutions do not consider both transient and steady-state performances. Such negligence is a critical obstacle for validating and verifying these solutions (LALANDA; MCCANN; DIACONESCU, 2013). Therefore, exploiting and deepening the formalism of autonomic computing and self-adaptive systems could help design more robust, reliable, and scalable caching strategies.

Self-adaptive software embodies a closed-loop mechanism. This loop typically involves four main activities: monitoring, analysis, planning, and execution, with the addition of a shared knowledge-base. As depicted in Figure 2.4, sensors collect data from the managed system. The feedback cycle starts with the monitoring of relevant data that reflect the current state of the system. The values of measurable properties of a system's states are referred to as *variables*. Next, the autonomic manager analyzes the collected data, structuring, and reasoning about the raw data. Then, decisions must be planned about how to adapt the system to reach a desirable state. Finally, to implement the decision, the autonomic manager must execute it using available effectors. In the core of this loop, there is a knowledge base that keeps the necessary information about the managed entities and their operations. This closed-loop is also known as an adaptation or

Figure 2.4: The Closed Control Loop of Self-adaptive Systems.



feedback loop, and this reference model is also referred to as *MAPE-K* by the autonomic computing community (HUEBSCHER; MCCANN, 2008).

In the context of caching, the monitoring activity is responsible for collecting relevant data that affects caching decisions, strategies and policies, such as application behavior and cache statistics. Then, the collected data is analyzed to identify unfavorable caching decisions and their possible causes. Based on such analysis, a decision is planned, taking into consideration past actions, application state, and cache configuration. Next, the planned action is executed, which can entail changes in caching decisions. Finally, the control loop restarts, considering updated information and conditions of the caching and the results of past executions. According to this proposed adaptation loop, there are six key properties that could describe an adaptive system, as described below.

Monitored Data Monitored data are measurable input parameters from which the adaptive mechanism can infer the status of the managed system.

Analysis An analysis process should be employed in order to characterize the system performance over a sampling period. Such characterization can be based on a single monitored data or a synthesis of a set of input parameters, such as a utility function, or even a complex model built through machine learning approaches.

Behavior The behavior represents how the adaptive component acts over the managed resource or system. It can be reactive, i.e. responding to observed events after its occurs, or proactive, i.e. taking actions in advance, before events have a chance to occur.

Operation The operation corresponds to how the algorithm can process data and take actions. Online operation implies it can process its inputs piece-by-piece in a serial fashion, without requiring the entire input to produce outputs. In contrast, offline operation requires the whole monitored data (or at least a large portion) to output a solution for the problem.

Decision Adaptive systems implement a transfer function that takes estimated values as input and gives the amount of change (if needed) in the expected properties as output.

Goal Goals are expected properties consist of one or more attributes of the managed system that can be manipulated to apply the necessary adaptations. They summarize the desired system performance or behavior regarding input parameters.

To illustrate, let us consider an adaptive replacement mechanism that uses a utility function based on recency and frequency of cached items for determining a hit estimation, which is used as a criterion to replace cached items in situations when the cache achieves its full capacity, and there is a new entry to add. If the hit estimation of a cached item is below a certain value (e.g., 50%), then it becomes a good candidate to be evicted. In this example, recency and frequency are the monitored data, given that the autonomic manager cannot control them. The hit estimation (i.e. the utility function) is the analysis, because it summarizes monitored data, and is used for deliberating changes. Such algorithm acts reactively and online, because it takes actions only when the maximum cache size is achieved, and is able to process monitored data and produce decisions at runtime, without requiring an asynchronous analysis to characterize the monitored data. Finally, the threshold value of 50% is the goal, while the hit threshold is the decision, which is used to guide changing actions in the cache.

Given that we introduced application-level caching and provided the background needed to understand this work, we now proceed to the presentation of our effort to provide means for reducing the challenges that application-level caching imposes for developers.

3 A QUALITATIVE STUDY OF APPLICATION-LEVEL CACHING

Although application-level caching is commonly being adopted, it is typically implemented in an *ad hoc* way, and there are no available practical guidelines for developers to appropriately design, implement and maintain caching in their applications. To find caching best practices, developers can make use of conventional wisdom, consult development blogs, or simply search online for tips. Nevertheless, this empirical knowledge is unsupported by concrete data or theoretical foundations that demonstrate its effectiveness in practice. Given this scenario, an essential step towards approaches that can support developers to reach a good trade-off between cache development and application performance is to *understand*, *extract*, *structure* and *document* the application-level caching knowledge implicit and spread in existing web applications.

Thus, we performed a qualitative study to identify patterns and guidelines to support developers while designing, implementing and maintaining application-level caching in their applications. Previous qualitative studies have been conducted to gain a greater understanding of the behavior of developers during specific software development tasks (ROBILLARD; COELHO; MURPHY, 2004; SILLITO; MURPHY; De Volder, 2008; NADI et al., 2015; BORSTLER; PAECH, 2016; NAMOUN et al., 2016). Moreover, there is work that focused on proposing software development guidelines to help software developers while developing applications (JORGENSEN, 2005; JURISTO; MORENO; SANCHEZ-SEGURA, 2007; CARVAJAL et al., 2013). Our research is similar in nature, but focuses on the investigation of a development challenge that has not been previously addressed.

Besides helping developers while handling cache, the novel findings of this study provide insights to propose solutions to raise the abstraction level of caching or automate caching tasks, providing a better experience with caching for developers. The results of this study are thus informative and serve as practical guidance for the development of application-level caching, given that there is neither off-the-shelf solutions or systematic ways of designing, implementing, and maintaining it. It is important to note, however, that qualitative research is aimed at gaining a deep understanding of a particular target of study, rather than providing a general description of a large sample of a population.

3.1 Study Design

We performed a qualitative study to understand how application-level caching is implemented in existing software systems that rely on this kind of caching. We provide details of our study in next sections, starting by discussing the study approach in Section 3.1.1. We present our goal and research questions in Section 3.1.2, and then describe the study procedure in Section 3.1.3. We introduce the target systems of our study in Section 3.1.4, to later proceed to the analysis and interpretation of our obtained results.

3.1.1 Study Approach

We chose a qualitative method rather than quantitative in order to take a holistic and comprehensive understanding of caching practices adopted by developers, exploring their inner experiences in application development.

Our study was designed based on comparative and interactive principles of *grounded theory* (GLASER, 1992). The purpose of grounded theory is to construct theory grounded in data by identifying general concepts, develop theoretical explanations that reach beyond the known, and offer new insights into the area of study. The systematic procedures of grounded theory enable qualitative researchers to generate ideas. In turn, these ideas can be later studied and verified through traditional quantitative forms of research.

3.1.2 Goal and Research Questions

Our primary objective while performing this study is to provide guidance to developers when adopting application-level caching in their applications. This study aims to provide such guidance using an *application-centric approach*, i.e. by identifying caching solutions that developers usually apply in their applications, taking into account application details. The study was guided by the framework proposed by Basili et al. (BASILI; SELBY; HUTCHENS, 1986; BASILI; CALDIERA; ROMBACH, 1994). The paradigm includes the goal-question-metric (GQM) template, which was adopted to define the goal of the study, the research questions to be answered to achieve the goal and metrics for answering these questions.

Although the GQM approach has been extensively used as a way to structure and

Table 3.1: Goal Definition (GQM Template).

Definition Element	Our Study Goal
Motivation	To identify patterns of application-level caching adopted by developers and understand what kinds of caching implementations and decisions can be automatically inferred,
Purpose	characterize and evaluate
Object	application-level caching-related design and implementation
Perspective	from a perspective of the researcher
Domain: web-based applications	as they are implemented in the source code and described in issues of web-based applications
Scope	in the context of 10 software projects, obtained from open-source repositories and software companies.

design empirical studies, it is focused primarily on *quantitative* studies. Given that our study is qualitative rather than quantitative, instead of metrics we specify an evaluation approach with sub-questions, which are used in a similar way as metrics, in the sense of being a way to guide the analysis and answer the mapped questions. This helped us to select the means of achieving our study goal. The description of the study, following the GQM template, is presented in Table 3.1.

To achieve our goal, we investigated different application-level caching concerns, which are associated with the three key research questions presented below.

RQ1. What and when is data cached at the application level?

RQ2. How is application-level caching implemented?

RQ3. Which design choices were made to maintain the application-level cache efficient?

Determining the cacheable content and the right moment of caching or clearing the cache content are a developer's responsibility and might not be trivial in complex applications, motivating RQ1. In this research question, we aim to identify *what* data is selected to be cached, and the criteria used to detect such cacheable data. Furthermore, this question also explores *when* data should be evicted, as well as constraints, consistency conditions, and the rationale for all these choices.

In addition to design issues, as shown in Figure 2.3(a), the cache system and the underlying source of data are not aware of each other, and the application must implement ways to interact with the cache. Therefore, our goal with RQ2 is to characterize patterns of *how* this implementation occurs in the application code; for example, ways to assigning names to cached values, performing lookups, and keeping the cache up to date.

Finally, determining maintenance strategies to manage efficiently *where* data is placed, such as replacement policies or size of the cache, requires additional knowledge and reasoning from application developers. Nevertheless, there are no foundations to make this choice adequately. Therefore, RQ3 complements the analysis above, questioning how application specificities are leveraged to provide the desired performance to the cache system.

By answering these three central questions, we can extract patterns and guidelines for caching design, implementation and maintenance, and also the context in which each pattern can be adopted.

3.1.3 Procedure

As previously described, this study is mainly based on development information of web applications. To investigate different caching constructs, we followed a set of steps to perform our qualitative study: (i) selection of a set of suitable web applications; (ii) specification of a set of questions to guide us in the data analysis and (iii) analysis of each web application using the specified questions. The collected data consists of six different sources of information, explained as follows.

Information about the application. Our goal is to identify caching patterns or decisions, which possibly depend on the application domain. Therefore, we collected general details of the applications to characterize them. The collected application data is (i) its description; (ii) programming languages and technologies involved; and (iii) size of the application in terms of the number of lines of code.

Source code. Application source code is our core source of information. Since we focus on application-level caching, our analysis is concentrated in the core of the application (i.e. the business logic), which is where the caching logic is typically implemented.

Code comments. Given that caching is an orthogonal concern in the application, unrelated to the business logic, but interleaved with its code, code comments are often used to describe the reasons behind caching implementation and decisions.

Issues. An issue can represent a software bug, a project task, a help-desk ticket, a leave request form, or even user messages about the project in general. Usually, changes in the code are due to registered issues. Thus, implementation and design decisions are better explained by associated issues in issue platforms, such as GitHub Issue Tracker, JIRA, and Bugzilla.

Developer documentation. In general, developer documentation consists of useful resources, guides and reference material for developers to learn how the application was implemented. Thus, the available documentation about the project was also collected.

Developers. In addition to the manual inspection of the above data, we asked developers to which we had access about caching-related implementation, decisions, challenges, and problems.

To guide the extraction of the information needed from the above data for our qualitative analysis, we first derived a small set of sub-questions for each research question, based on a broad analysis of our qualitative data. Such sub-questions conveyed characteristics that should be observed and evaluated while looking for answers to the main research questions, i.e. sub-questions were used to extract data. In addition, based on the observations made during the analysis, new sub-questions emerged as well as the existent sub-questions were refined. Table 3.2 depicts the derived sub-questions, which serve as a checklist while analyzing our data. Results presented in Section 3.2 are derived from answers to these questions.

We followed the analytical process of *coding* in our analysis (GLASER, 1992), which makes it easier to search the data and identify patterns. This process combines the data for themes, ideas and categories, labeling similar passages of text with a code label, allowing them to be easily retrieved at a later stage for further comparison and analysis. We used what we learned from the analysis to adapt our evaluation approach and observation protocols. Insights we had while coding the data and clustering the codes were captured in memos. There are three coding phases in classical grounded theory: open coding, selective coding, and theoretical coding. Open coding generates concepts from the data that will become the building blocks for the theory (GLASER, 1992). The process of doing grounded theory is based on a concept indicator model of constant comparisons of incidents or indicators to incidents (GLASER; STRAUSS, 1967). Indicators are actual data, such as behavioral actions and events observed or described in documents and interviews. In this case, an indicator may be an architectural style or design pattern adopted to implement the cache, a data structure, a class, a control flow logic, a comment, a discussion in the issues platform, a paragraph in the documentation, or any other evidence we can get from the data being analyzed.

By using grounded theory, we aimed to construct a well-integrated set of hypotheses that explain how the concepts operated. Thus, the selective coding involves identifying the core category that best explains how study data refers to a large portion of the variation

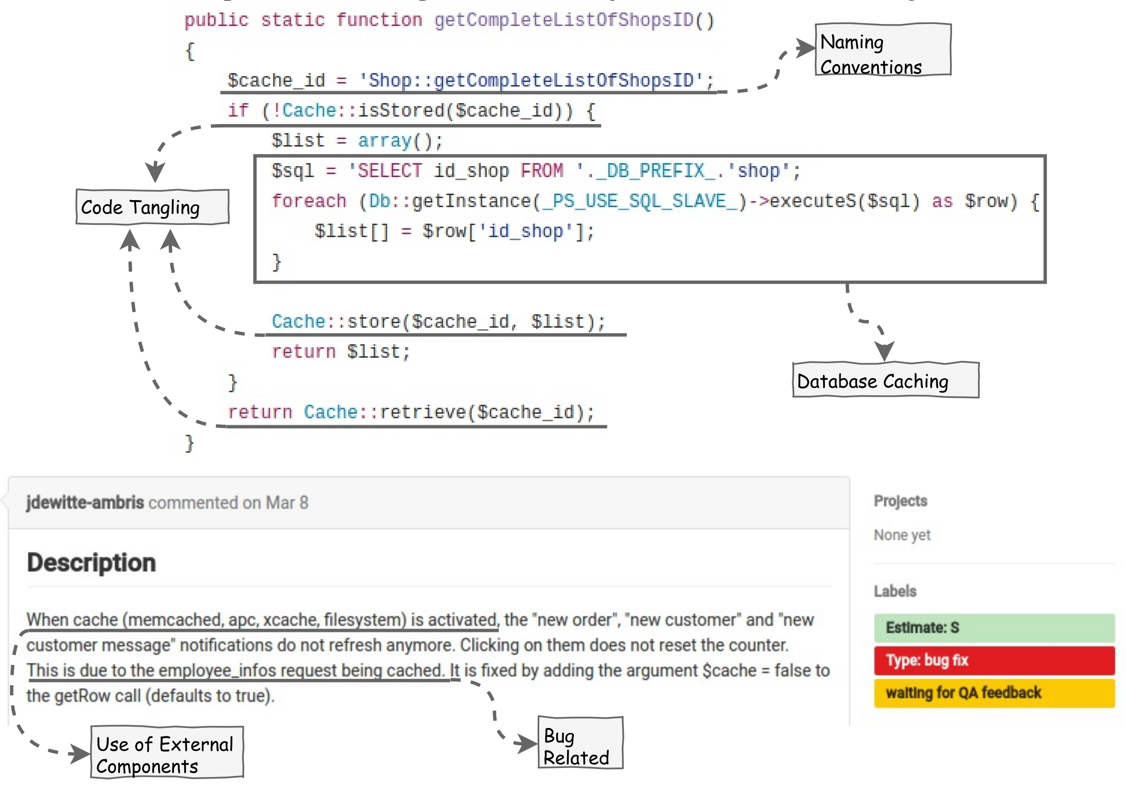
Table 3.2: Evaluation Approach (Sub-questions).

#	Sub-question	RQ1. What and when is data cached at the application level?	RQ2. How is application-level caching implemented?	RQ3. Which design choices were made to maintain the application-level cache efficient?
1	What are the motivations to employ cache?	X		
2	What are the typical use scenarios?	X		
3	Where and when data is cached?	X		
4	What are the constraints related to data cached?	X		
5	What are the selection criteria adopted to detect cacheable content?	X		
6	Do developers adopt a pattern to decide which content should be cached?	X		
7	What kinds of bottlenecks are addressed by developers?	X		
8	What motivated the need for explicit caching manipulation?	X		
9	What is the granularity of the cached objects?	X		
10	What is the importance of the cached data to the application?	X		
11	Is there a relationship between the data cached and the application domain?	X		
12	How much memory does the cached data consume?	X		
13	What data is most frequently accessed?	X		
14	How often is the cached data going to be used and changed?	X		
15	What data is expensive?	X		
16	What data depends on user sessions?	X		
17	How up to date does the data need to be?	X	X	
18	How is consistency assurance implemented? Why was it chosen?	X	X	
19	Where and when is consistency assurance employed?	X	X	
20	Which kind of operation/behavior affects cache consistency?	X	X	
21	Do developers employ any technique to ease caching implementation, such as design patterns, third-party libraries or aspects?		X	
22	How is the caching logic mixed with the application code?	X	X	
23	Is this extra cache logic tested?		X	
24	What is the required format to cache?		X	X
25	How are objects translated to the cache?		X	X
26	How are names (keys) defined for cached objects?	X	X	
27	Do developers use another caching layer besides application-level?		X	
28	Is any transparent or automatic caching component being used?		X	X
29	Do developers rely on automatic caching components?		X	X
30	Is it necessary to explicitly manipulate a caching component that should be automatically managed?		X	X
31	Which application layers are more likely to have caching logic?	X	X	
32	Do developers perform analysis to measure cache efficiency?			X
33	What is the replacement policy adopted? Why was it chosen?			X
34	What is the size of the cache? Why was it defined?			X
35	What are the default parameter values?			X
36	Do developers use configurations different from to default?		X	X
37	Do developers take into account application-specific information when defining maintenance strategies?		X	X

in a pattern and is considered the primary concern or problem related to the study, integrating closely related concepts. Finally, theoretical codes conceptualize how the codes may relate to each other as hypotheses to be incorporated into the theory (GLASER, 1992).

Figure 3.1 provides an example of such analytical process and illustrates how we collected the different evidences. First, we assigned concepts to pieces of extracted text (open coding), each representing application-level caching characteristics. Figure 3.1 exemplifies different codes identified during the analysis of an application. For instance, *Code Tangling* is created from the observation of cache-related implementation spread all over the application base code (underlined). Then, for each new concept, we verified whether they are connected somehow with existing ones, in order to generate categories (selective coding). Thus, the name assigned to a particular category aims at representing, at a higher abstraction level, all concepts related to it. Regarding the *Code Tangling* example, if *Code Scattered* were found afterwards, we could establish a relationship between the former and the latter to create a category, given that both are related to lack of separation of concerns.

Figure 3.1: Example of the Analytical Process of *Coding*.



```
public static function getCompleteListOfShopsID()
{
    $cache_id = 'Shop::getCompleteListOfShopsID';
    if (!Cache::isStored($cache_id)) {
        $list = array();
        $sql = 'SELECT id_shop FROM '._DB_PREFIX_.'shop';
        foreach (Db::getInstance(_PS_USE_SQL_SLAVE_)->executeS($sql) as $row) {
            $list[] = $row['id_shop'];
        }

        Cache::store($cache_id, $list);
        return $list;
    }
    return Cache::retrieve($cache_id);
}
```

jdewitte-ambris commented on Mar 8

Description

When cache (memcached, apc, xcache, filesystem) is activated, the "new order", "new customer" and "new customer message" notifications do not refresh anymore. Clicking on them does not reset the counter. This is due to the employee_infos request being cached. It is fixed by adding the argument $cache = false to the getRow call (defaults to true).

Following the introduced phases of data analysis, we performed mainly a subjective analysis of the data, collecting: (i) typical caching design, implementation and maintenance strategies; (ii) motivations, challenges and problems behind caching, and (iii) characteristics of caching decisions. These collected data were evaluated to conceptualize how the open codes were related to each other as a set of hypotheses in accounting for resolving the primary concern. Furthermore, we also made a broad analysis of the target systems in order to investigate how application-level caching was conceived in them. All the research phases were performed manually, as the collected data (most of them expressed in natural language) analysis is associated with the interpretation of caching approaches.

3.1.4 Target Systems

In order to investigate different aspects of caching, it was important to select systems that make extensive use of application-level caching. To obtain applications that employ application-level caching, we searched through open-source repositories, from which information can be easily retrieved for our study. Based on text search mech-

Table 3.3: Target Systems of our Study.

Project Name	**Domain**	**Language**	**KLOC**	**Analysis Order**
S1	Market trend analysis	Ruby	21	#8
Pencilblue	CMS and blogging platform	JavaScript	33	#2
Spree	e-Commerce	Ruby	50	#7
Shopizer	e-Commerce	Java	57	#3
Discourse	Platform for community discussion	Ruby	88	#5
OpenCart	e-Commerce	PHP	123	#4
OpenMRS core	Patient-based medical record system	Java	127	#6
ownCloud core	File sharing for online collaboration and storage	PHP	193	#10
PrestaShop	e-Commerce	PHP	245	#1
Open edX	Online courses platform	Python	250	#9

anisms, we assessed how many occurrences of cache implementation and issues were present in applications to ensure they would contribute to the study. The more caching-related implementation and issues, the better, so that the rationale behind choices regarding application-level caching could be extracted. Moreover, we managed to obtain commercial applications with partner companies interested in the results of this study.

Therefore, we analyzed systems with a broad range of characteristics. Aiming at reducing the influence of particular software features on the results, we selected systems of different sizes (from 21 KLOC to 250 KLOC, without comments and blank lines), written in different programming languages, adopting different frameworks and architectural styles, and spanning different domains. We studied ten systems in total, of which nine are open-source software projects, and one is from the industry. Due to intellectual property constraints, we will refer to the commercial system as S1. Table 3.3 summarizes the general characteristics of each target system.

The open-source applications were selected from GitHub, the widely known common code hosting service. We selected GitHub projects that match the following criteria: (i) projects with some popularity (at least 350 stars); (ii) projects containing application-level caching implementation and issues (at least 50 occurrences of cache-related aspects); (iii) projects written in different programming languages; and (iv) projects of different domains. The first criterion indicates that projects are interesting and were possibly evolved by people other than the initial developers. The second ensures that selected projects would present caching-related implementation and issues, which would contribute more to the study. The inclusion of applications was not restricted to any particular technology or programming language given that the study is focused on identifying language-independent caching patterns and guidelines expressed in the source code. Furthermore, we found applications that use cache in other architectural (e.g. database systems) or infrastructure (e.g. proxy caching) levels. However, these caching approaches are not handled directly by the developers within the application; consequently, such applications do not fit the purposes of our study. The commercial system was selected by convenience. We selected the applications that better satisfied criteria (i) and (ii), without repeating programming languages and domains. In order to minimize bias, applications were analyzed according to a randomly generated order, which is presented in the last column of Table 3.3.

It is important to highlight that our goal is to capture the *state-of-practice* of application-level caching, by characterizing and deriving caching patterns. Consequently, we assume that the investigated applications—in which caching was introduced in previous releases and was already debugged—have adequate application-level implementations, i.e. we did not evaluate the quality of any caching implementation.

3.2 Analysis and Results

This section details the results of our study and their analysis, according to the research questions we aim to answer. Our collected data consist mainly of source code and issues (expressed in natural language) and, as these are qualitative data, we have undertaken a subjective analysis of the application-level caching (hereafter referred to simply as "caching") aspects represented in the target systems. Note that we labeled some findings with "*Evidence X*," so that we can later refer to them to support the guidelines we derived from this analysis.

Before addressing each of our research questions, we show an objective analysis of the impact of caching in the investigated applications by identifying all code and issues related to them. This gives a broad sense of how caching is implemented in target systems.

3.2.1 Caching in Target Systems

To investigate how caching is present in target systems, we examined the number of files in which caching logic is implemented, the number of LOC associated specifically with caching (without comments and blank lines), and the number of issues related to it. This analysis is shown in Table 3.4, in the columns *#Cache Files*, *Cache LOC* and *#Cache Issues*, respectively. This table also gives an overview of further information of each

Table 3.4: Characteristics of Target Systems.

Project Name	#Cache Files	Cache LOC	#Cache Issues	Doc.	Dev.
S1	30 (4.72%)	446 (2.12%)	NA	No	Yes
Pencilblue	16 (7.04%)	526 (1.59%)	26 (4.99%)	No	No
Spree	27 (2.58%)	422 (0.84%)	205 (2.92%)	Yes	No
Shopizer	31 (4.20%)	1725 (3.02%)	0 (0%)	No	No
Discourse	53 (3.10%)	1190 (1.35%)	48 (1.34%)	Yes	No
OpenCart	44 (4.23%)	462 (0.37%)	77 (1.97%)	Yes	No
OpenMRS core	22 (2.06%)	350 (0.27%)	19 (1.11%)	Yes	No
ownCloud core	240 (10.52%)	4344 (2.24%)	670 (2.99%)	Yes	No
PrestaShop	54 (4.78%)	2727 (1.11%)	95 (1.96%)	Yes	No
Open edX	198 (10.76%)	4693 (1.87%)	333 (2.95%)	No	No

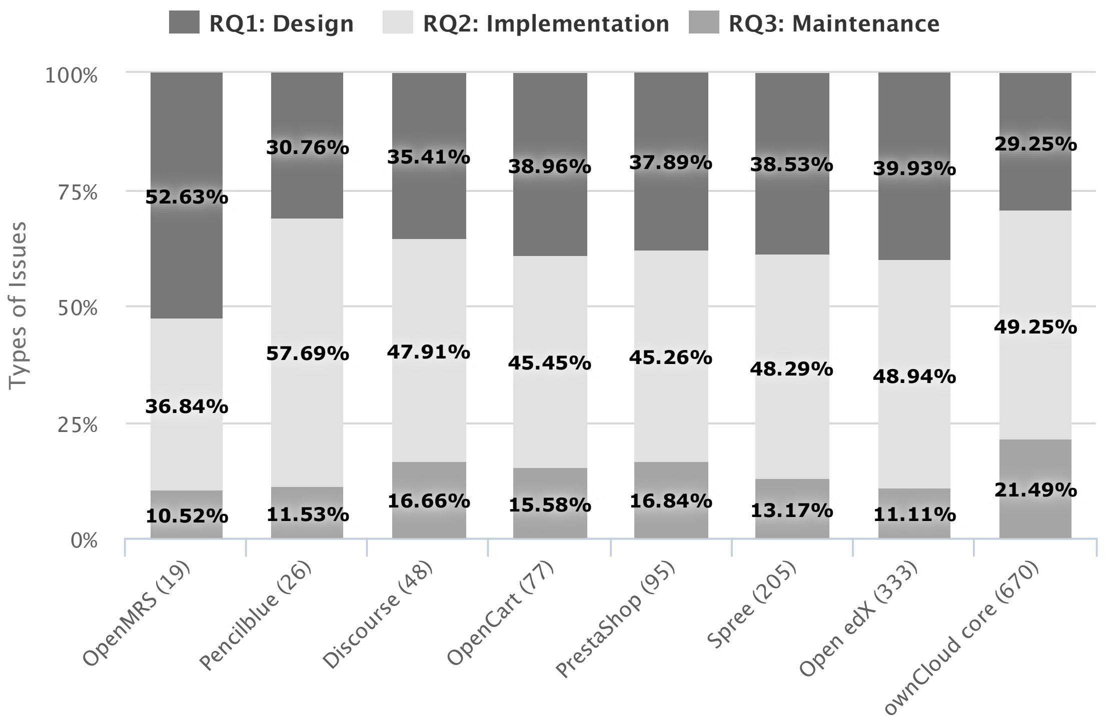
application to which we had access. They are some form of documentation and access to developers. It is important to note that available documentation about caching, in most cases, was limited to an abstract or general description of how caching was adopted. This documentation was available in some open source systems, and we only had access to developers of our system from the industry.

Based on data presented in Table 3.4, we can observe a significant amount of lines of code dedicated to implementing caching, ranging from 0.27% to 3.02%. It shows the importance of the caching in the project, considering that caching is a solution for a non-functional requirement (i.e. scalability and performance). Furthermore, caching logic is presented in a substantial amount of files, from 2.06% to 10.76%, which indicates the caching nature of being spread all over the application.

Moreover, we observed that all analyzed web applications did not adopt caching since their conception. As they increased in size, usage and performance analysis were performed and led to requests for improvements. Thus, developers had to refactor data access logic to encapsulate cache data into the proper application-level object, which is a task that can be very time-consuming, error-prone and, consequently, a common source of bugs. As result, we found a significant number of issues specifically related to caching, achieving the maximum of 4.99% of the *Pencilblue* issues (*Evidence 15*).

We performed an analysis of the issues available in issue platforms to investigate the primary sources of cache-related problems, typically bugs, in the applications. Based on user messages, code reviews and commit messages that are described in the issues, we classified them into three different categories, which follow the main topic of our research questions: *Design* (e.g. changes in the selection of content to be put in the cache), *Implementation* (e.g. bugs in the implementation) and *Maintenance* (e.g. performance tests, adjustments in replacement algorithms or size limits of the cache). Results of the performed analysis are presented in Figure 3.2, in which applications are ordered ascending

Figure 3.2: Classification of Caching Issues by Topic.



by the number of cache-related issues, which is shown next to each application name. Only open source projects with issues related to caching are detailed in this figure.

As can be seen in Figure 3.2, caching implementation and associated design decisions are much more discussed and revised by developers than maintenance decisions. Caching implementation, which is spread in the code and involves the choice for appropriate locations to add and remove elements from the cache, is error-prone and can compromise code legibility. Consequently, many issues are associated with bug fixes, technological details and code refactorings. Moreover, despite being less frequent, caching design is time-consuming and challenging, given that it requires understanding of the application behavior, as well as limitations, conditions and restrictions of content being cached. In applications analyzed, the mean (M) and standard deviation (SD) values of cache-related implementation issues are M = 47.45% and SD = 5.76%, while design issues achieve M = 37.92% and SD = 7.11%. Finally, because fine-grained configurations require empirical analysis such as cache profiling, and there is little evidence that this was performed in investigated applications, maintenance decisions often result in the choice for default settings. Consequently, a lower number of issues is associated with such decisions, specifically M = 14.61% and SD = 3.44%.

After this broad analysis of our target systems, we further analyze them focusing on our research questions. For each research question, we detail coding results and explore the answers achieved while analyzing emerged codes.

3.2.2 RQ1: What and when is data cached at the application level?

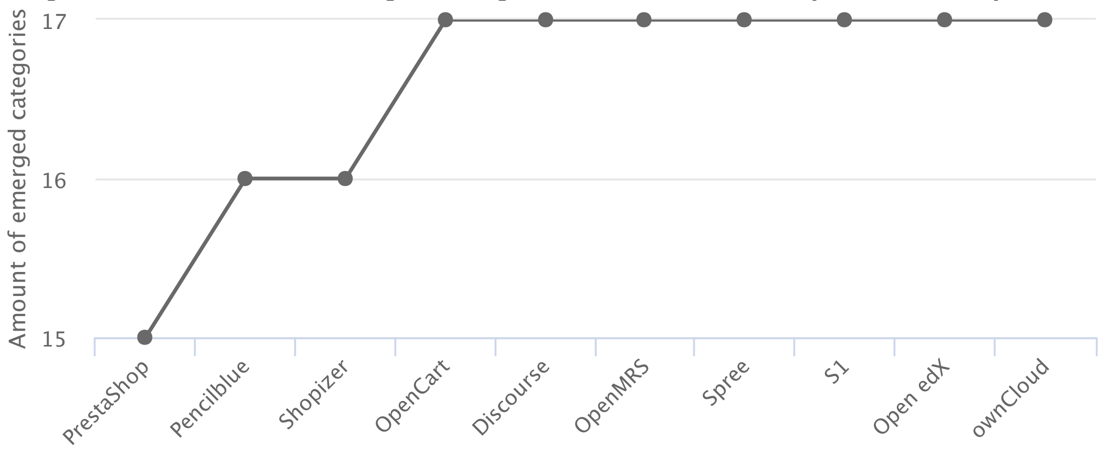
In this question, we focus on going beyond the facts exposed by source code and analyze the reasoning behind caching decisions such as what and why data is cached or evicted, when and where caching is done, and what are the conditions and constraints involved with this. Therefore, issues, code comments, and documentation about the cache of applications were the primary source of information to find answers to this question, because they convey (in natural language) the rationale behind implementation decisions.

With the analysis of all provided information about application-level caching, we gathered and categorized it in an exhaustive way, observing fine-grained details. As said, we followed the principles of grounded theory, and its first phase consists of open coding. During the coding phase, the questions presented in Table 3.2 were used to drive the process. However, the inspection of our qualitative data was not limited to answering these questions. As result, new questions emerged and were incorporated to the list of questions (or those existing were refined). As result, we identified 72 initial categories. Following the process, these categories were not predefined and were created and reviewed during the open coding process.

These initial categories that emerged during open coding were posteriorly analyzed and conceptualized, focusing on identifying relationships between them as hypotheses and grouping categories that may be theoretically coded as causal and associated with degrees. The open codes were directly used to derive more abstract categories during the theoretical coding. In the end, we identified 17 concepts, which better convey the implicit knowledge about application-level caching design, implementation, and maintenance acquired from the applications. After the analysis of the fourth application, codes stabilized, as shown in Figure 3.3.

The theoretical categories are described in Table 3.5, which presents their description, an original piece of content (example of evidence) and sources of data classified in each category. Moreover, categories are also shown in Figure 3.5(a) with the associated number of occurrences in the applications analyzed and classified according to each research question. Figure 3.5(b) shows the percentage of contribution of each application to the categories emerged from the study. These figures also present categories identified

Figure 3.3: Amount of Emerged Categories based on the Analysis of Each System.



in *RQ2* (described in Table 3.6) and *RQ3* (described in Table 3.7), which are discussed in the following sections. Acronyms used in Figures 3.5(a) and 3.5(b) are introduced in Tables 3.5, 3.6, and 3.7.

We observed in eight from the ten analyzed applications that developers indicated that they had *uncertainty in cache design*, with respect to deciding what data should be cached and the right moment to do it, causing missed caching opportunities. There are comments in the source code and issues about whether to cache some specific data or not, showing that the connection between observed bottlenecks in the application and opportunities for caching is not straightforward and requires a deeper analysis (*Evidence 1*).

Therefore, selection criteria based on common sense or past experiences are initial assumptions to ease such decisions. Although these criteria are usually unjustified, they guide developers while selecting content. We observed in 90% of the applications the definition of selection criteria to make the distinction of cacheable from uncacheable content easier. These criteria were observed in *explanations of cache design choices* in comments, issues, and documentation. We identified common criteria used to determine whether to cache or not a specific content, which are: (i) content change frequency (*Evidence 2*), (ii) content usage frequency (*Evidence 3*), (iii) content shareability (*Evidence 4*), (iv) content retrieval complexity (*Evidence 5*), (v) content size (*Evidence 6*), and (vi) size of the cache (*Evidence 7*).

Regarding design choices, we observed common practices. The first is associated with the lack of a specific approach to cache data. To process a client request, application components of distinct layers and other systems (databases, web services, and others) are

Table 3.5: Analysis of RQ1: Emerged Design Categories.

Category (Acronym)	**Description**	**Example of Evidence**	**Evidence Sources**				
			SC	**COM**	**IS**	**DOC**	**DEV**
Uncertainty in Cache Design (UCD)	Indication, by developers, of uncertainty regarding cacheable content.	A code comment before an expensive operation: *TODO cache the global properties to speed this up??*		X	X		X
Explanations of Cache Design Choices (EDC)	Explanations provided by developers regarding cache design choices, such as specific criteria to determine cacheable content.	A code comment detailing the motivation for caching: *Fetch all dashboard data. This can be an expensive request when the cached data has expired, and the server must collect the data again.*	X	X	X	X	
Multiple Caching Solutions (MCS)	Occurrences of multiple caching solutions, which can involve different third-party components or different application layers. As a consequence, the same content can be cached at different places in varying forms.	A user announcement on issue platform: *Caching has now landed: Fragment caching for each product; Fragment caching for the lists of products in home/index and products/index; Caching in the ProductsController, using expires_in which caches for 15 minutes.*	X	X	X	X	
Caching in Application Boundaries (CAB)	Occurrences of caching content being added to the cache because they retrieve data from frameworks, libraries, or external applications.	A sentence in the documentation: *Most stores spend much time serving up the same pages over and over again. [...] In such cases, a caching solution may be appropriate and can improve server performance by bypassing time-consuming operations such as database access.*	X	X	X	X	X
Ensuring Consistency (ENC)	Design of some kind of consistency approach such as expiration policies and invalidation, preventing stale data.	A user message on issue platform detailing the invalidation approach adopted: *Ideally we cache until the topic changes (a new post is added, or a post changed) [...] less ideally we cache for N minutes*	X	X	X	X	X
Complex Cache Design (CCD)	Indication that a cache design choice is difficult to understand, confusing developers and requiring detailed comments.	A user message on issue platform: *i can see that it may speed up the query responses, but is the saved time substantial enough to be worth the effort?*			X		X
Choice for Simple Cache Design Solutions (CSD)	Indication that developers selected simple solutions for cache rather than complex ones, such as defining a time-to-live (TTL) instead of implementing manual invalidation, possibly in order to balance design effort and caching gains.	A code snippet defining a default setting: *<defaultCache maxElementsInMemory="1000" eternal="false" timeToIdleSeconds="60" timeToLiveSeconds="0" overflowToDisk="false" diskPersistent="false"/>*	X	X	X		X

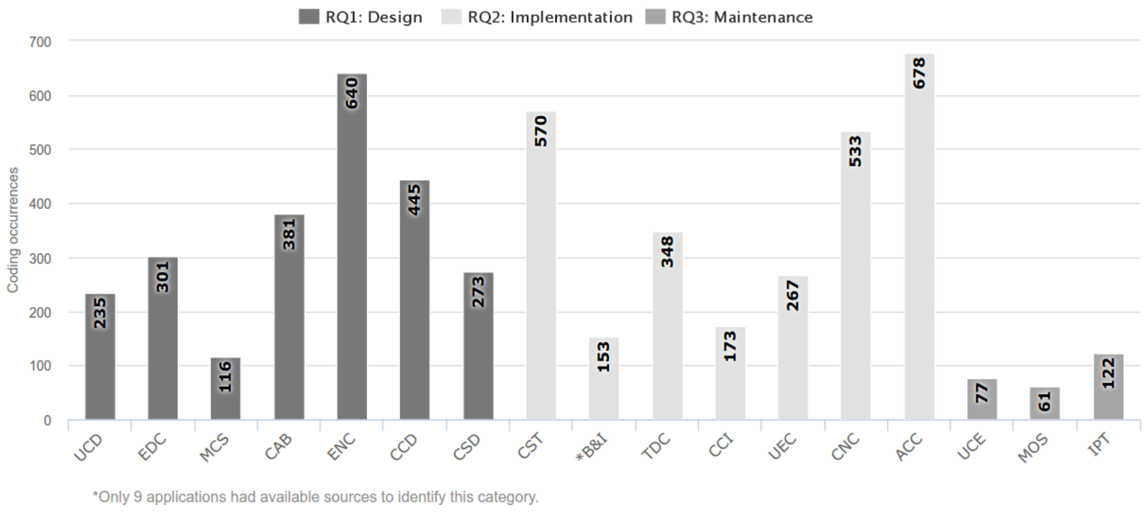
Labels of Evidence Sources: **SC**-Source Code (without comments); **COM**-Code Comments; **IS**-Issues; **DOC**-Documentation; **DEV**-Developers.

invoked, and each interaction results in data transformation, which is likely cacheable. Due to this, nine analyzed applications present *multiple caching solutions* by not specifying cacheable layers, components or data, and employing cache wherever can potentially provide performance and scalability benefits (*Evidence 8*), no matter which is the application layer, component or data (*Evidence 9*). As a consequence, the same content can be cached at different places, from the database to controllers, in varying forms such as query results, page fragments or lists of objects (*Evidence 10*).

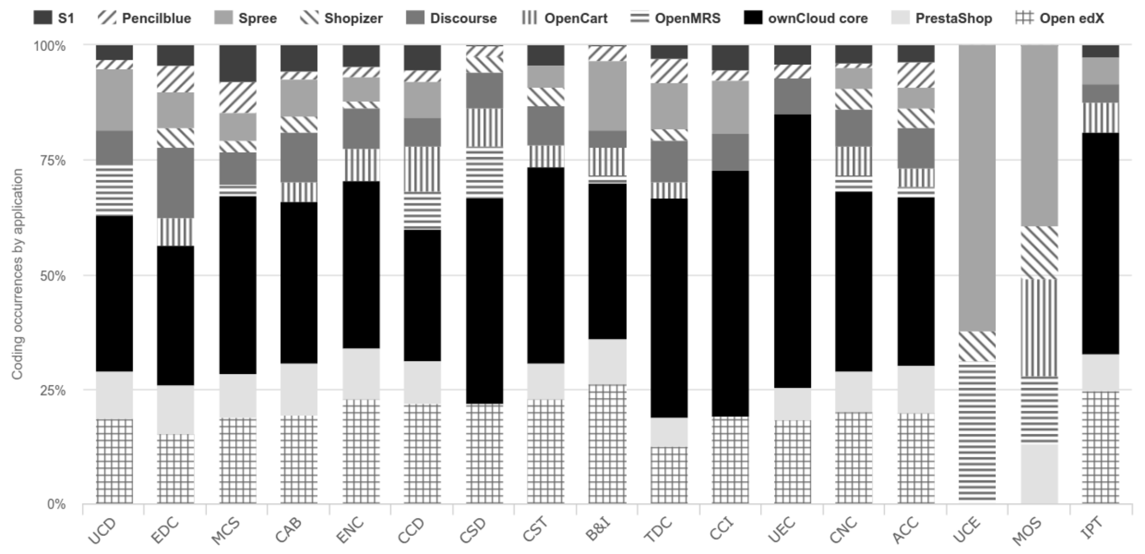
The second design choice is associated with the concern of reducing the communication latency between the application and other systems, which increases the overall response time of a user request. Therefore, we noticed *caching in application boundaries* in nine of the analyzed applications, addressing remote server calls and requests to web services, database queries, and loads dependent on file systems, which are common bottlenecks (*Evidence 11*).

Despite choosing where and what to cache, cached values are valid only as long as the sources do not change, and when sources change, a consistency policy should be employed to ensure that the application is not serving stale data. Therefore, in nine analyzed applications, there are indications that developers demand an effort to design *consistency approaches*, reasoning about the lifetime of cached data, as well as eviction

Figure 3.4: Categories Emerged from the Study.



(a) Coding Occurrences.



(b) Coding Percentage by Application.

conditions and constraints. We identified common approaches to keep consistency, which are: (i) a less efficient and straightforward approach is to invalidate cached values based on mapping actions that change its dependencies and, after a change is invoked, invalidate cached values and recompute them at the next request; and (ii) the use of a less intrusive alternative such as a time-to-live (TTL) or a replacement approach, which require a certain level of staleness (*Evidence 12*).

All these caching design options may become complex and difficult to understand. Indeed, identifying caching opportunities and ensuring consistency can add much code and may not be trivial to implement and understand. Due to the nature of application-level caching, such logic is spread all over the system. We noticed in 90% of the applications the presence of pieces of evidence that *caching design achieves a high level of complexity*, requiring many comments explaining even less complex parts and issues related to the lack of understanding of caching decisions (*Evidence 13*).

Due to the increasing demand and complexity involved with caching, developers try to decrease cache-related effort by *adopting simple design solutions*. We observed in seven of the ten analyzed applications design choices such as managing consistency based on expiration time, keeping default parameters, selecting data without any criteria, or even adopting external solutions, which do not claim extra reasoning, time, and modifications to the code (*Evidence 14*).

We discussed our findings regarding cache design decisions and we next discuss observations made associated with how to implement design decisions.

3.2.3 RQ2: How is application-level caching implemented?

To understand how developers integrate application-level caching logic in their applications, we analyzed the cache implementations from two perspectives. The first consists of examining the explicit caching logic present in the application code, focusing on analyzing where the caching logic is placed, for what this logic is responsible, and when it is executed. The second evaluates the integration and communication between the application and an external caching component, which is usually used to relieve the burden on developers, easing cache implementation, raising abstraction levels, or even taking full control of caching. For this question, the most valuable data comes from source code and comments, which express implementation details. The theoretical categories referring to *RQ2* are described in Table 3.6 and shown in Figure 3.4.

We observed in eight from the ten analyzed applications that they present *code scattering and tangling*, on caching logic, causing low cohesion and high coupling in the code. Caching control code, responsible for caching data in particular places, was spread all over the base code, being invoked when application requests were processed. Consequently, there is a lack of separation of concerns, leading to increased maintenance effort (*Evidence 15*).

A possible cause for this is that caching was not part of all the applications since their conception. Applications were developed and, as they evolved, usage and performance problem reports led to requests for response time improvements. Thus, developers had to refactor existing business logic to include caching aspects. Interleaved caching

Table 3.6: Analysis of RQ2: Emerged Implementation Categories.

Category (Acronym)	**Description**	**Example of Evidence**	**Evidence Sources**				
			SC	**COM**	**IS**	**DOC**	**DEV**
Code Scattering and Tangling (CST)	Presence of caching logic spread all over the application, indicating a lack of separation of concerns.	A developer quote: *At first glance, the cache code hinders the understanding of the business logic. Also, the cache logic itself it is not easy to get.*	X	X	X		X
Bugs and Issues (B&I)	Presence of bugs and problems due to caching reporting, for example, increased client response time, reduced throughput, increased server resource utilization.	A bug report on an issue platform: *When you import categories with a parent category which does not exist, it prevents from duplicate it because of the cache.*			X		
Technical Debt Concerns (TDC)	Indication of extra effort spent by developers to build an extensible and well designed caching component.	A user message on an issue platform: *my concern with this implementation is that it can lead to some confusing situations. [...] I think the caching needs to be more granular*	X	X	X	X	X
Complex Caching Implementation (CCI)	Presence of complex constructs such as batch processing and asynchronous communication, which require extra effort and reasoning from developers to be implemented.	A user message on an issue platform: *Can you give me some tips on how to safely implement an async step to save this? [...] I could have the end user polling and refreshing against that as needed.*	X	X	X	X	
Use of External Components (UEC)	Indication of use of third-party caching solutions to help the implementation, raising the level of abstraction of some caching aspects.	A sentence in documentation: *We use Redis [a third-party solution] as a cache and for transient data.*	X	X	X	X	X
Complex Naming Conventions (CNC)	Choice for complex keys of caching content, causing developers to spend time and effort to elaborate and understand such keys.	A code snippet: *cache_id = ’objectmodel_’ . $entity_defs[’classname’] . ’_’ . (int)$id . ’_’ . (int)$id_shop . ’_’ . (int)$id_lang;*	X	X	X		
Additional Caching Code (ACC)	Code implemented to support the caching logic, such as implemented caching tests, logic to monitor cache statistics and additional interfaces to support available caching providers.	A code snippet exposing a test of caching logic: *it "can set and get false values when return cache nil" do @store.set :test, false expect(@store.get(:test)).to be false end*	X	X	X		

Labels of Evidence Sources: **SC**-Source Code (without comments); **COM**-Code Comments; **IS**-Issues; **DOC**-Documentation; **DEV**-Developers.

logic can cause not only increased maintenance effort but also be a source of bugs—in eight (from nine, given that we had no access to an issue tracker of one of the applications) analyzed applications, there are issues associated with *bugs due to caching* (*Evidence 16*).

Although this problem is present in the code, there are indications that developers know about it and express they are willing to improve the provided solution. In order to reduce the impact of an infrastructure component to the system business logic, we identified cases where there are suggestions to design more extensible classes and modules, refactoring and reducing cache-related code, and reusing components (*Evidence 17*). This acknowledgment of *technical debt* was observed in 90% of the applications.

Regarding implementation choices, we observed common practices. The first is associated with how to name cached data. In order to use in-memory caching solutions, there is no prescribed way to organize data. Typically, unique names are assigned to each cached content, thus leading to a key-value model—and this was the case in all investigated applications. Given that cache stores lots of data, the set of possible names must be large; otherwise, two names (keys) can conflict with each other and, thus stale (or even entirely wrong) data can be retrieved from the cache. Consequently, *complex naming conventions* were adopted (*Evidence 18*).

Therefore, in all applications, there are indications, in the form of source code, comments, and issues, that developers demand an effort to understand and improve cache keys, reasoning about alternative better ways to organize and identify content. We identified common content properties used to build cache keys, which are: (i) content *ids*, (ii) method signatures, and (iii) a tag-based identification, in which the type of content, the class, the module or hierarchy are used (*Evidence 19*).

The second implementation choice is associated with the concern of not increasing the throughput due to the introduction of communication between the cache and the application. Due to this, six of our applications make *use of non-trivial programming techniques* to ensure a high performance caching implementation, taking advantage of every improvement opportunity (*Evidence 20*). Solutions related to this include processing caching requests in the background, batching caching calls together, or even implementing a coarse-grained caching service that allows a single logical operation to be performed by using a single round trip. These prevent blocking client requests when a possibly large caching processing is being performed, e.g. a caching warm up or update (*Evidence 21*).

Given that implementing cache is challenging, we noticed that in six applications developers made *use of supporting libraries and frameworks*. This was done to prevent adding much cache-related code to the base code, because such components raise the abstraction level of caching, providing some ready-to-use features (*Evidence 22*). Examples of such external components are distributed cache systems, e.g. Redis[1] and Memcached, and libraries that cache content locally, e.g. Spring Caching[2], EhCache, Infinispan[3], Caffeine[4] and Rails low-level caching[5].

Such supporting libraries and frameworks not only provide partial ready-to-use features but also reduce the amount of additional effort required to guarantee that the cache is working. We observed in all applications that the cache includes *code dedicated to test, debug and configure cache components*, which can be expensive in some scenarios (*Evidence 23*).

We next discuss observations made associated with how the cache is maintained and tuned.

[1]<https://redis.io/>
[2]<https://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html>
[3]<http://infinispan.org/>
[4]<https://github.com/ben-manes/caffeine>
[5]<http://edgeguides.rubyonrails.org/caching_with_rails.html#low-level-caching>

Table 3.7: Analysis of RQ3: Emerged Maintenance Categories.

Category (Acronym)	**Description**	**Example of Evidence**	**Evidence Sources**				
			SC	**COM**	**IS**	**DOC**	**DEV**
Uncertainty in External Cache Component (UCE)	Indication of uncertainty in the behavior of an external cache component, when the applications rely on it instead of handling the cache manually.	A code comment: *Is hibernate taking care of caching and not hitting the db every time? (hopefully it is)*	X	X	X	X	
Maintenance of Cache Size (MOS)	Indication of choices made in order to control the cache size, such as definitions of eviction approaches to fit cache size limitations.	A user message on issue platform: *Be sure the local cache will not grow out of control (especially during big operations like product import)*	X	X	X		
Improvements based on Performance Tests (IPT)	Indication that improvements were made based on performance tests.	An user message on issue platform comparing two request log traces, with and without cache: *My results were something like: 1) Using rows: 30ms, 5mb RAM (peak usage); 2) Fetching all at once: 3ms, 5.75MB RAM*			X		X

Labels of Evidence Sources: **SC**-Source Code (without comments); **COM**-Code Comments; **IS**-Issues; **DOC**-Documentation; **DEV**-Developers.

3.2.4 RQ3: Which design choices were made to maintain the application-level cache efficient?

After designing and implementing the cache, maintenance issues are still open. Initial assumptions concerning caching design can become invalid as changes occur in the application domain, workload characteristics or access patterns, thus leading to a performance decay. Therefore, in order to keep the cache efficient, it is necessary to constantly tune cache settings, at least in between releases. Maintenance decisions involve: (i) detecting improvement opportunities in the cache design, (ii) dealing with specificities of the cache deployment scheme (e.g. local or remote, shared or dedicated, in memory or on disk storage), (iii) determining the appropriate size of the cache, (iv) defining how many objects are allowed to be cached at same time, and (v) what should be done in case of the cache is full. The theoretical categories identified while investigating how developers approach these issues and maintain cache efficient are described in Table 3.7 and shown in Figure 3.4.

As mentioned in Section 3.2.2, we observed that developers often rely on external components, which can raise the level of caching abstraction, providing some cache-related components ready-to-use. The most common tasks delegated to external components are associated with cache space allocation and management, since maintaining cache manually through lists or hashes inside the application involves issues such as dealing with concurrency, key management and size limits, which may not be trivial.

Furthermore, in cases where updates in the base code are not an option (due to time or technical restrictions), a transparent and automatic caching component can provide fast results. These solutions address layers before and after application boundaries, and require only a few adaptations to the application needs (*Evidence 24*).

However, we observed in three analyzed applications an *uncertainty regarding the quality, in terms of performance, of a transparent component* (*Evidence 25*). In fact, transparent caching solutions are out of the application control and are built and maintained by a third party community. Although they provide ways to developers to manipulate and observe their behavior, it may become a problem when requests for improvements force developers to customize the behavior of the built-in cache component, thus creating a new and particular behavior for the component, which is supposed to be generic and easy-to-use. All this manipulation should also be tested and evaluated, leading to the generation of test cases to assert the reliability of the external component being used.

The use of transparent and automatic caching solutions relieve developers also from two maintenance tasks. First, while it is true that the larger the cache size, the better the performance, very large caches are unrealistic. While cost is the primary reason for limiting the size of a cache, efficient invalidation approaches play a key role as well. We noticed in five of our analyzed applications indications that developers explicitly *deal with size limitations* (*Evidence 26*), by defining a specific size of the cache, replacement policies (*Evidence 27*) and the number of manageable objects allowed to be cached.

The second maintenance task refers to cache performance improvements. We observed in seven of the ten applications indications that developers *perform tests*, comparing the application behavior with and without a particular caching approach (*Evidence 28*). However, while focusing on tuning all the aspects of caching, developers perform trivial analyses, which are usually based on comparing logs or execution of several requests in a row and measurement of response times. These tests are inappropriately reported and documented, becoming irreproducible. As result, these tests can lead to false-positive improvements, based on poor and biased conclusions (*Evidence 29*).

3.3 Threats to validity

We now analyze the possible threats to the validity of this study, and how we mitigated them. Researcher bias is a typical threat to qualitative research because the results are subject to the researcher interpretation of the data. In our study, prior knowledge and the fact that just one researcher was responsible for conducting the coding might have influenced results. In order to avoid this, we followed a systematic analysis of our data, and conclusions are all founded on these data. Moreover, cross-checks were performed using our different sources of evidence.

Another source of bias in this study is the process adopted to select applications and the number of applications analyzed. We mitigated this problem by specifying criteria (described in Section 3.1.4) that allowed us to select systems with different characteristics to be part of this study. Moreover, we observed that our codes and patterns emerged after analyzing part of our systems and, in the remaining ones, we identified only recurrences of codes and patterns. Furthermore, given that the goal of qualitative research is to understand a particular event, rather than providing a general description of a large sample of a population, selecting few representative applications is sufficient for our study.

The existence of such plateau reached after the analysis of the fourth application also gives evidence that the results may be widely applied to other web applications. However, we acknowledge that the selected applications may have influenced the results of this study as well as the order in which they were analyzed, thus being an external threat to the validity of this study. We mitigated this threat by selecting a sample of applications with different domains, programming languages, technologies, sizes and business models, and the use of a random order to analyze them.

Although caching is associated with performance concerns and requirements, the focus of this study is on software engineering concerns of application-level caching, approaching design, implementation, and maintenance from a developer point of view. It is important to highlight that our goal is to capture the state-of-practice of application-level caching, by characterizing and deriving caching patterns. Consequently, we assume that the investigated applications have adequate caching implementations given that caching was introduced in previous releases and was already debugged and improved by developers, i.e. we assume that the design and implementation are all efficient; otherwise, there would be open issues in issue trackers to be resolved by developers. Therefore, another threat in our research is related to the possibility that we observed inadequately designed and implemented caching. Furthermore, the quality or performance improvement of design and implementation of application-level caching could be better explored in future work.

Finally, most of the web applications satisfy initial performance and scalability requirements using traditional forms of caching. The need for application-level caching usually appears after the application is released, with an increasing user demand. Due to this, all applications selected in our study did not adopt application-level caching since their conception—no application implemented with application-level caching in its first release was found. Consequently, it is not possible to assess if the overhead of adding cache to an application in later stages is higher than doing so during its initial development. For all analyzed systems, performed complexity or usage analysis led to requests for improvements, as systems became larger. Consequently, developers had to refactor the application to insert caching logic. This is thus another external threat to validity, given that our results cannot be generalized to systems that use application-level caching since they are conceived.

3.4 Final Remarks

We observed a higher number of categories, which are classes of observations made based on the analysis of these applications, associated with caching design and implementation than those associated with maintenance. Furthermore, the number of occurrences of each category, design and implementation categories also have higher numbers. This phenomenon was expected since the most representative portion of our qualitative data consists of source code and issues. Moreover, simple maintenance tasks and configurations are already executed and provided by external components, being commonly adopted. However, the number of occurrences do not reflect the importance of a category.

The high number of occurrences related to *ensuring consistency* refers to the expiration process of cached content, which requires extra reasoning from developers, because they should track which changes cause data content to become outdated, and be aware for how long the cache can provide stale data, in case the data source has been updated. In fact, consistency approaches have been widely investigated (PORTS et al., 2010; GUPTA; ZELDOVICH; MADDEN, 2011), and the typical way of dealing with it is to analyze data dependency, from which conditions and constraints for consistency can be derived.

Many emerged categories, such as *Bugs and Issues*, *Technical Debt Concerns* and *Complex Design and Implementation*, indicate the additional burden placed on developers while designing and implementing application-level caching, which can lead to a significant amount of time and effort added to software projects. Therefore, ideally, opportunities for caching should be identified as early as possible during the application design, avoiding the problems of employing cache only in late stages of the development cycle or even after it is in production, as an emergency solution. Furthermore, we observed that faster results can be achieved when a caching component is not implemented from scratch, but when libraries and frameworks are adopted to support cache implementation.

From all analyzed applications, we observed that none of them use a proactive approach to cache content. Content is always cached after it requested (i.e. reactive approach) and, as a consequence, the first request always results in a cache miss. Due to this, prefetching techniques can be used in order to populate the cache and prevent misses by predicting and caching data that will potentially be requested in the future. It can be based on heuristics, usage observations or even with the use of complex prediction algorithms (PODLIPNIG; BÖSZÖRMENYI, 2003; DOMÈNECH et al., 2006; ALI; SHAMSUDDIN; ISMAIL, 2011; PAL; JAIN, 2014). However, the design and implementation of a reactive cache component already requires significant effort and reasoning to be properly done, and a proactive approach increases the complexity of the caching solution even more.

Based on our analysis, we identified application-level caching decisions and behaviors adopted by developers. Our findings allowed us to propose a set of guidelines and patterns for the development of a caching approach, which are described in next chapter.

4 GUIDELINES AND PATTERNS

The study described in the previous chapter allowed us to understand how developers deal with application-level caching in their applications, by explaining design, implementation and maintenance choices. Our findings and observations were used as foundation to the provision of practical guidance for developers with respect to caching.

4.1 Guidelines

In this section, we thus introduce guidelines, which are general rules to be followed by developers while developing application-level caching, and patterns, which are structured reusable solutions to recurring problems in cache design, implementation or maintenance. For each guideline, we indicate evidences that support it. Both guidelines and patterns derived from our study are classified into categories, explained below. Note that there are patterns associated with the proposed guidelines.

Design. Support to design decisions associated with application-level caching.

Implementation. Support to implementation issues of application-level caching, by providing solutions and guidance at the code level.

Maintenance. Support to performance analysis and improvement of application-level caching.

4.1.1 Design Guidelines

Evaluate different abstraction levels to cache. (*Evidence 9*) and (*Evidence 11*) It is important to cache data where it reduces the most processing power and round trips, choosing locations that support the lifetime needed for the cached items, despite where it is located in the application. Different levels of caching provide different behavior, and possibilities must be analyzed. For instance, caching in the model or database level offers higher hit ratios, while caching in presentation layer can reduce the application processing overhead significantly in the application in case of a hit. However, in the latter case, hit ratios are in general lower. It is possible to cache data at various layers of an application, according to the following layer-by-layer considerations.

Controller layer. Caching data at the controller layer should be considered when data needs to be frequently displayed to the user and is not cached on a per-user basis. At this level, controllers usually work by serving parameterized content, which can be used as an identifier in the cache. For example, if a list of states is presented to the user, the application can load these once from the database and then cache them, according to the parameters passed in the first request.

Business or service layer. Caching data at the business layer should be considered if an application needs to process requests from the presentation layer or when the data cannot be efficiently retrieved from the database or another service. It can be implemented by using hash tables, library or framework. However, at this level, a large amount of data tends to be manipulated and caching it can consume more memory and leads to memory bottlenecks.

Model or database layer. At the model or database layer, a large amount of data can be cached, for lengthy periods. It is useful to cache data from a database when it demands a long time to process queries, avoiding unnecessary round-trips.

Stack caching layers. (*Evidence 9*) and (*Evidence 10*) It is reasonable to say that the more data cached, the lower the chance of being hit without any content already loaded. Caching might be at the client, proxy server, inside the application in presentation, business, and model logics, or database. Although the same data may be cached in multiple locations, when the cache expires in one of them, the application will not be hit with an entirely uncached content, avoiding processing and network round trips. However, it is important to keep in mind that caching layers imply a range of responsibilities, such as consistency conditions and constraints, and extra code and configuration. Due to this, it is important to consider many caching layers but, at the same time, achieve a good trade-off between caching benefits and implementation effort.

Separate dynamic from static data. (*Evidence 3*) and (*Evidence 2*) Content can be distinguished in static, dynamic, and user-specific. By partitioning the content, it is easier to select portions of the data to cache.

Evaluate application boundaries. (*Evidence 10*) and (*Evidence 11*) Communication between application and external components is a common bottleneck and, consequently, an opportunity for caching. Consider caching for database queries, remote server calls and requests to web services, which are made across a network.

Specify selection criteria. (*Evidence 1*), (*Evidence 2*), (*Evidence 3*), (*Evidence 5*), (*Evidence 6*) and (*Evidence 7*) Selecting the right data to cache involves a great reasoning effort given that data manipulated by web applications range in dynamicity, from being completely static to changing constantly. To optimize this selection process, there are four primary selection criteria used by developers while detecting cacheable content, which should be used in decisions regarding whether to cache. These criteria are described below, ordered according to their importance; i.e. the higher the influence level, the earlier it is presented.

Data change frequency. Developers should seek for data that have some degree of stability, i.e. those that are more used than changed. Even if data are volatile and change in time intervals, caching still brings a benefit. This is the first factor to be considered since caching volatile data implies the implementation of consistency mechanisms, which is not trivial and requires an extra effort and reasoning from developers. In short, the cost of consistency approaches cannot be higher than the benefit of caching. Besides, when stale data is not a critical issue, an approach of weak consistency can be employed, such as time-to-live (TTL) eviction, where data is expired after a time in cache, regardless of possible changes.

Data usage frequency. Frequent requests, operations, queries and shared content (accessed by multiple users) must be identified, focusing on recomputation avoidance. Even if some processing can be fast enough at a glance, it can potentially become a bottleneck when being invoked many times. Despite being frequently used, user-specific data cannot be shared and may not bring the benefit of caching, being usually left out of the cache.

Data retrieval complexity. Data that is expensive to retrieve, compute, or render, regardless of its dynamicity, is always considered a good caching opportunity.

Size of the data. The size of the content being cached should be considered when using size-limited caches. In this case, an adequate trade-off between popularity (hits) and size of the items must be achieved. Keeping small popular items in the cache tends to optimize hit-ratio; however, a hit in a large item may be more beneficial for an application than many hits on small items. At the same time, filling the cache with few large items may turn the cache performance dependent on a good replacement policy.

Evaluate staleness and lifetime of cached data. (*Evidence 2*), (*Evidence 3*) and (*Evidence 12*) Every piece of cached data is already potentially stale, it is important to rethink the degree of integrity and potential staleness that the application can compromise for increased performance and scalability. Many cache implementations adopted an expiration policy to invalidate cached data based on a timeout since weak consistency is easier than defining a hard-to-maintain, but more robust, invalidation process. In short, developers must ensure that the expiration policy matches the pattern of access to applications that use the data, which is based on determining how often the cached information is allowed to be outdated, and relaxing freshness when possible.

Avoid caching per-user data. (*Evidence 4*) and (*Evidence 26*) When the application is running on the server-side, it is recommended to avoid caching per-user data unless the user base is small and the total size of the cached data does not require an excessive amount of memory; otherwise, it can cause a memory bottleneck. However, if users tend to be active for a while and then go away again, caching per-user data for short-time periods may be an appropriate approach. For instance, a search engine that caches query results by each user, so that it can page through results efficiently. However, in the case of modern web applications which have business logic on the client-side, personalized content should be considered for caching.

Avoid caching volatile data. (*Evidence 2*) and (*Evidence 3*) Data should be cached when it is frequently used and is not continually changing. Developers should remember that caching is most effective for relatively stable data, or data that is frequently read. Caching volatile data, which is required to be accurate or updated in real time, should be avoided.

Do not discard small improvements. (*Evidence 3*), (*Evidence 8*) and (*Evidence 9*) The user perceived latency is reduced by any caching solution employed. This means that even not obvious scenarios should be target of caching, i.e. it is not true that solely data that is frequently used and expensive to retrieve or create should considered for caching. Furthermore, data that is expensive to retrieve and is modified on a periodic basis can still improve performance and scalability when properly managed. Caching data even for a few seconds can make a large difference in high volume sites. If the data is handled more often than it is updated, it is also a candidate for caching.

4.1.2 Implementation Guidelines

Keep the cache API simple. (*Evidence 15*), (*Evidence 17*), (*Evidence 20*) and (*Evidence 22*) Caching logic tends to be spread all over the application, and a good solution should be employed to avoid writing messy code at the cost of high maintenance efforts.

Define naming conventions. (*Evidence 19*) and (*Evidence 18*) To define appropriate names for cached data, it is important to assign a name that is related to its context, the data itself, and the caching location. It can provide two direct benefits: (a) prevention of key conflicts, and (b) guidance of cache actions such as updates and deletes of stale data in case of changes in the source of information.

Perform cache actions asynchronously. (*Evidence 21*) For large caches, it is adequate to load the cache asynchronously with a separate thread or batch process. Moreover, when an expired cache content is requested, it needs to be repopulated and doing so synchronously affects response time and blocks the request processing thread.

Do not use cache as data storage. (*Evidence 27*) An application can modify data held in a cache, but the cache should be considered as a transient data store that can disappear at any time. Therefore, developers should not save valuable data only in the cache, but keep the information where it should be as well, minimizing the chances of losing data if the cache unexpectedly becomes unavailable.

4.1.3 Maintenance Guidelines

Perform measurements. (*Evidence 1*), (*Evidence 8*), (*Evidence 26*), (*Evidence 28*) and (*Evidence 29*) Caching is an optimization technique and, as any optimization, it is important perform measurements before making substantial changes, given that not all application performance and scalability problems can be solved with caching. Furthermore, if unnecessarily employed, caching can eventually decrease performance rather than improve it.

Document and report measurements. (*Evidence 29*) To compare and reproduce performance tests employed, it is important to document the setup used to perform the application analysis. It includes modules enabled, particular configurations and hardware settings.

Consider using supporting libraries and frameworks. (*Evidence 22*) and (*Evidence 24*) Supporting libraries and frameworks can raise the level of abstraction of cache implementation and provide useful features. In addition, they can scale up in a much easier and faster way than application-wide solutions.

Tune default configurations. (*Evidence 22*), (*Evidence 25*), (*Evidence 26*) and (*Evidence 27*) Default configurations provided by external components serve as a start point. However, because they are generic and may not fit the application specificities,

Table 4.1: Caching Guidelines Derived from the Study.

Guideline	**Associated Evidence**	**ID**
Evaluate different abstraction levels to cache	(*Evidence 9*) and (*Evidence 11*)	DG-01
Stack caching layers	(*Evidence 9*) and (*Evidence 10*)	DG-02
Separate dynamic from static data	(*Evidence 3*) and (*Evidence 2*)	DG-03
Evaluate application boundaries	(*Evidence 10*) and (*Evidence 11*)	DG-04
Specify selection criteria	(*Evidence 1*), (*Evidence 2*), (*Evidence 3*), (*Evidence 5*), (*Evidence 6*) and (*Evidence 7*)	DG-05
Evaluate staleness and lifetime of cached data	(*Evidence 2*), (*Evidence 3*) and (*Evidence 12*)	DG-06
Avoid caching per-user data	(*Evidence 4*) and (*Evidence 26*)	DG-07
Avoid caching volatile data	(*Evidence 2*) and (*Evidence 3*)	DG-08
Do not discard small improvements	(*Evidence 3*), (*Evidence 8*) and (*Evidence 9*)	DG-09
Keep the cache API simple	(*Evidence 15*), (*Evidence 17*), (*Evidence 20*) and (*Evidence 22*)	IG-01
Define naming conventions	(*Evidence 19*) and (*Evidence 18*)	IG-02
Perform cache actions asynchronously	(*Evidence 21*)	IG-03
Do not use cache as data storage	(*Evidence 27*)	IG-04
Perform measurements	(*Evidence 1*), (*Evidence 8*), (*Evidence 26*), (*Evidence 28*) and (*Evidence 29*)	MG-01
Document and report measurements	(*Evidence 29*)	MG-02
Consider using supporting libraries and frameworks	(*Evidence 22*) and (*Evidence 24*)	MG-03
Tune default configurations	(*Evidence 22*), (*Evidence 25*), (*Evidence 26*) and (*Evidence 27*)	MG-04
Use of transparent caching components	(*Evidence 22*) and (*Evidence 25*)	MG-05

other configurations must be evaluated and possibly adopted.

Use of transparent caching components. (*Evidence 22*) and (*Evidence 25*) The use of transparent caching solutions to address bottlenecks outside the application boundaries such as databases, final HTML pages or fragments, and static assets can provide fast results. These solutions do not explore application specificities, but can still provide performance benefits for typical usage scenarios.

All the proposed guidelines are summarized in Table 4.1 with the respective name, associated evidence and identification.

4.2 Patterns

Based on our study, we derived caching patterns, which can be used by developers to help them design, implement and manage cache. They can be used in combination with our guidelines. Moreover, we identified the components these patterns must have, which comprise a template for a caching pattern catalog. These components are (i) a

Table 4.2: Caching Pattern Classification.

Pattern	**Classification**	**Intent**	**Associated Guidelines**
Asynchronous Loading	Implementation	Design a mediator to asynchronously deal with caching.	IG-03, MG-03
Cacheability	Design	Provide a reasoning process to decide whether to cache or not particular data.	DG-01, DG-03, DG-05, DG-07, DG-08, DG-09
Data Expiration	Design and Maintenance	Given cacheable content, provide a reasoning process to choose a consistency management approach based on data specificities.	DG-06, DG-08, MG-03, MG-04
Name Assignment	Implementation	Ensure a unique key and keep track of the content cached.	IG-02

classification, (ii) the pattern *intent*, (iii) the *problem* involved, (iv) the *solution* proposed, and (v) an *example*.

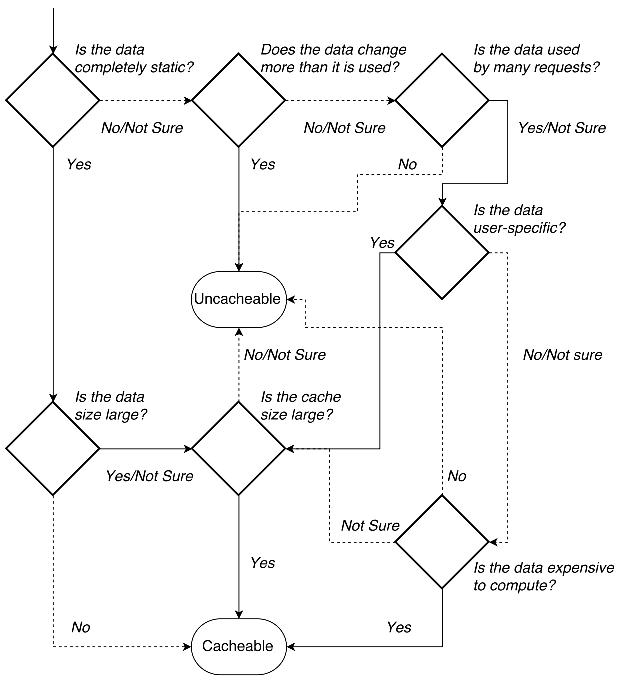
The proposed patterns are summarized in Table 4.2 along with the possible combination with guidelines. We limit ourselves to present in detail one of our patterns, namely the *Cacheability Pattern*, in Table 4.3, as an example. The complete description of the remaining patterns derived from our study is available in Appendix A.

4.3 Final Remarks

Based on the results of our qualitative study, guidelines and patterns were derived to provide guidance to developers to deal with application-level caching. The former are high-level directions to develop caching solutions, and also serve as a checklist of points that must be analyzed while designing and implementing a caching component. The latter capture the reasoning used by developers to build a caching component and integrating it with a web application, providing a systematic way to develop a caching solution. Regardless of the use of our guidelines or patterns (or any other reusable component), it is important to determine exactly whether a particular caching approach is performing adequately. Therefore, it is essential to perform an application profiling, by measuring performance both with and without caching.

Our guidelines and patterns can be used as foundation to design, implement and manage a caching component for a particular application as well as to develop an application-independent caching component or framework. Our approach, which is described next, involves the use of the guidance derived to raise the abstraction level of application-level caching and automate the reasoning captured in our patterns.

Table 4.3: Cacheability Pattern.

Classification: Design
Intent: provide a reasoning process to decide whether to cache or not particular data.
Problem: cache has limited size, so it is important to use the available space to cache data that maximizes the benefits provided to the application. Otherwise, it can end up reducing application performance instead of improving it, consuming more cache memory and at the same time suffering from cache misses, where the data is not getting served from cache but is fetched from the source.
Solution: even though there are many criteria that contribute for identifying the level of data cacheability, there is a subset that would confirm this decision regardless of the values of the other criteria. Changeability is the first criterion that should be analyzed while selecting cacheable data, then usage frequency, shareability, retrieval complexity, and cache properties should be considered. Figure 6 expresses a flowchart of the reasoning process to decide whether to cache data, based on the observation of data and cache properties. All criteria are tightly related to the application specificities and should be specified by the developer.  Is the data completely static? Does the data change more than it is used? Is the data used by many requests? No/Not Sure No/Not Sure Yes/Not Sure Yes Yes No Is the data user-specific? Yes Uncacheable No/Not Sure No/Not sure Is the data size large? Is the cache size large? Yes/Not Sure No Not Sure Yes Is the data expensive to compute? No Yes Cacheable *Rules of thumb:* (a) Despite being frequently used, user-specific data are not shareable and may not bring the benefit of caching, being usually avoided by developers. In this case, a specific session component is used to keep and retrieve user sessions. (b) If the data changes frequently, it should not be immediately discarded from cache. An evaluation of the performance benefits of caching against the cost of building the cache should be done. Caching frequently changing data can provide benefits if slightly stale data is allowed. (c) Expensive spots (when much processing is required to retrieve or create data) are bottlenecks that directly affect application performance and should be cached, even though it can increase complexity and responsibilities to deal with. Methods with high latency or that consists of a large call stack are some examples of this situation and opportunities for caching. In addition, we list content properties that should be avoided, which do not convey the influence factors in a good way and lead to problems such as cache trashing. (a) User-specific data. Avoid caching content that varies depending on the particularities of the request, unless weak consistency is acceptable. Otherwise, the cache can end up being fulfilled with small and less beneficial objects. As result, the caching component achieves its maximum capacity earlier and is flushed or replaced many times in a brief period, which is cache thrashing. (b) Highly time-sensitive data. Content that changes more than is used should not be cached given that it will not take advantage from caching. The cost of implementing and designing an efficient consistency policy may not be compensate. (c) Large-sized objects. Unless the size of the cache is large enough, do not cache large objects, it will probably result in a cache trashing problem, where the caching component is flushed or replaced many times in a short period.
Example: we list some typical scenarios where data should be cached and also give explanations based on the criteria presented. (a) Headlines. In most cases, headlines are shared by multiple users and updated infrequently. (b) Dashboards. Usually, much data need to be gathered across several application modules and manipulated to build a summarized information about the application. (c) Catalogs. Catalogs need to be updated at specific intervals, are shared across the application, and manipulated before sending the content to the client. (d) Metadata/configuration. Settings that do not frequently change, such as country/state lists, external resource addresses, logic/branching settings and tax definitions. (e) Historical datasets for reports. Costly to retrieve or create and does not need to change frequently.

5 AUTOMATED APPLICATION-LEVEL CACHING APPROACH

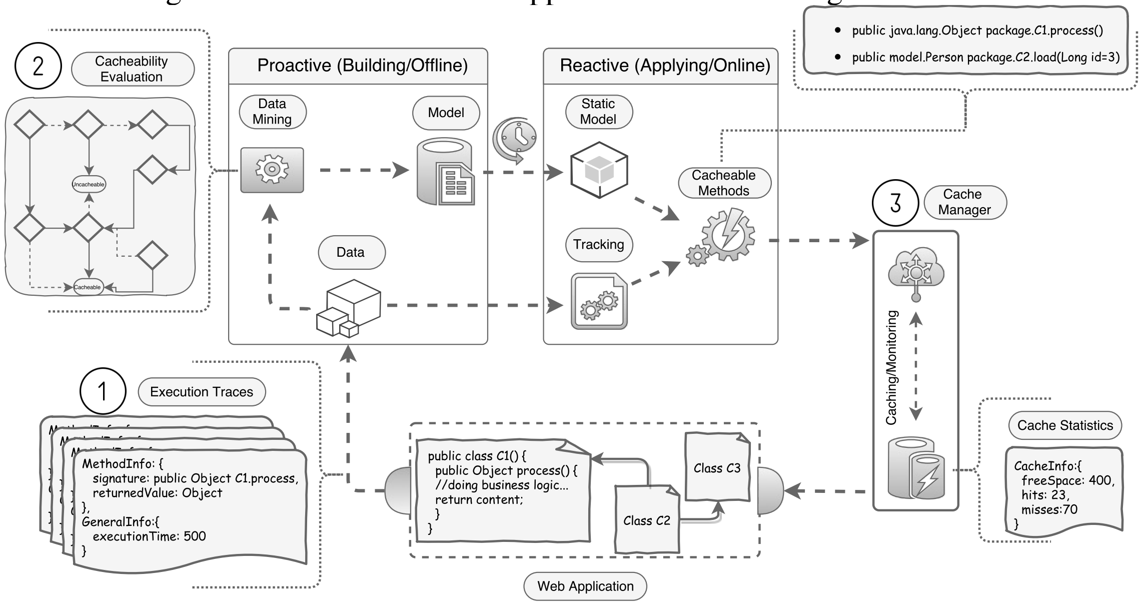
Because application-level caching is essentially a manual task, its design and implementation are time-consuming and error-prone. Moreover, because its implementation is often interleaved with the business logic, it decreases code understanding, thus being a common source of bugs. In this context there are a lot of open issues that could be addressed. As mentioned in Chapter 2, deciding the right content to cache and the best moment of caching are examples of challenging cache-related tasks, given that it depends on extensive knowledge of application specificities to be done efficiently; otherwise, caching may not improve application performance or even may lead to a performance decay (PORTS et al., 2010). Furthermore, developers must continuously inspect application performance and revise caching design choices, due to changing workload characteristics and access patterns. Consequently, initial caching choices may become obsolete (CHEN et al., 2016; RADHAKRISHNAN, 2004).

Thus, we propose a novel seamless and automated framework that manages the cache according to observations made by monitoring a web application at runtime, as well as detaches caching concerns from the application. Because it monitors the application at runtime, it is sensitive to the evolution of application workload and access patterns, thus being able to self-optimize caching decisions. Alternatively, it serves as a decision-support tool to help developers in the process of deciding what to cache, guiding them while manually implementing caching.

5.1 Approach Overview

Our approach consists of two complementary asynchronous parts: (a) a *proactive model building*, which monitors and analyzes the behavior of the application at runtime, taking into account information provided by an instance of a proposed meta-model that captures application specificities; and (b) a *reactive model applying*, responsible for caching method calls identified as caching opportunities based on traces of the application execution. Such parts involve performing different activities, which we conceptually explain next and, then, describe how they are operationalized within an implemented framework.

Figure 5.1: Overview of our Application-level Caching Framework.



```
• public java.lang.Object package.C1.process()
• public model.Person package.C2.load(Long id=3)
```

```
MethodInfo: {
  signature: public Object C1.process,
  returnedValue: Object
},
GeneralInfo:{
  executionTime: 500
}
```

```
public class C1() {
  public Object process() {
  //doing business logic...
  return content;
  }
}
```

```
CacheInfo:{
 freeSpace: 400,
 hits: 23,
 misses:70
}
```

5.2 Approach Activities

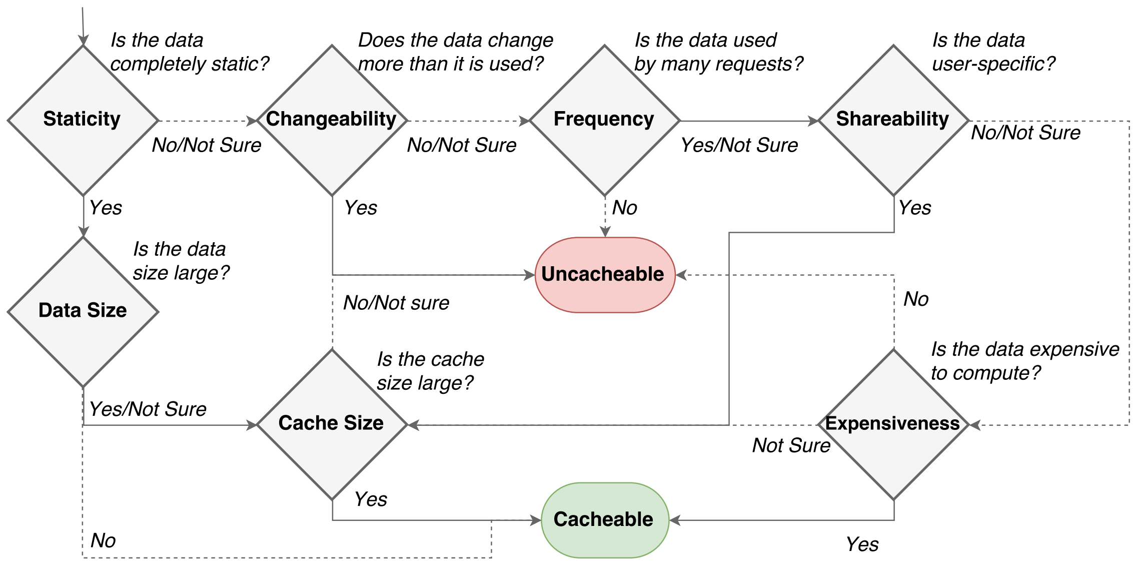
Figure 5.1 presents an overview of the dynamics of our approach, indicating the activities that comprise its running cycle. These activities are described next.

5.2.1 Data Tracking: Monitoring Execution Traces

To decide what to cache, we collect application execution traces at runtime. Such traces are associated with *method invocations or calls*, which are monitored during application execution. Related to this monitoring process, two issues must be addressed. First, we must specify what information should be recorded when invoking methods. Second, given that the monitored and recorded information may be a complex structure, it is essential to provide means of dealing with such complexity.

Concerning the first issue, we adopt a conservative approach. This means that the recorded information is the complete method call, consisting of the method identification (i.e. its signature), the values of all method inputs, its output (i.e. its returned value), and additional information, namely cost and user session. Each recorded call is a tuple $\langle s, P, r, c, u\rangle$, where s is the representation of the target of the call, $P = [p_1, \ldots, p_n]$ is a list of parameters of the call, r is the returned value, c is the cost of computing the method, and u is the user session associated with the method call. The cost can be time taken to

Figure 5.2: Cacheability Pattern with Associated Objective Evaluation.



the method be executed, memory consumed, or any other developer-specified resource.

The second issue involves dealing complex heap structures associated with method parameters and return values. These values are objects, which can have references to other objects that in turn can refer to other objects. As a consequence, recording entire object structures is unfeasible even for small-size web applications. This is addressed by transforming these objects into *hash* values.

5.2.2 Data Mining: Identifying Cacheable Methods

The previous activity provides us with information needed to identify methods to be cached, and this is achieved by mining such information. Essentially, this requires a statistical analysis of collected traces. In Chapter 3, we analyzed how developers deal with application-level caching. This work allowed us to derive a set of patterns that capture criteria used to make cache-related decisions. One of such patterns, namely the *Cacheability Pattern*, focuses on the selection of cacheable content. To guide the decision process, we specified the decision-making process as the pattern flowchart, presented in Figure 5.2, which is the same presented in Chapter 4, but in a different representation.

This reasoning model specifies a sequence of criteria to be analyzed and, by chaining different decisions regarding each criterion, an importance relationship among them is established. Content changeability is the first analyzed criterion, followed by usage frequency, shareability, cost to retrieve, and cache properties. To be automated, this model

Table 5.1: Objective Evaluation of the Cacheability Pattern Criteria.

Staticity. A method staticity is associated with how many times a method returns the same value when it receives the same parameters. Staticity is given by $staticity(m) = \|P_{Set}\|/\|Pr_{Set}\|$, where P_{Set} is the set of different lists P of parameter values received by a method m, and Pr_{Set} is the set of different tuples $\langle P, r\rangle$, where P is a list of parameters values and r is the returned value. A method m is said to be **completely static** if $staticity(m) = 1$.
Changeability. Static methods tend to achieve the highest hit ratio when cached. However, caching methods that, although are not static, often do not change can bring benefits. Therefore, the changeability of a method is the dual of staticity, i.e. $changeability(m) = 1 - staticity(m)$. In order to evaluate whether a method does not often change, we use as a reference value $\mu_{ch} + k \times \sigma_{ch}$, where μ_{ch} and σ_{ch} are the changeability mean and standard deviation, respectively, and k is a given number. The changeability criterion has an yes answer, when the method changeability is below the reference value, i.e. when it is k standard deviations below the changeability mean.
Frequency. A method frequency is associated with how many times a method is called, and this criterion is used in our approach also to assess whether the collected trace sample is large enough to make decisions. Therefore, we use a specified threshold to distinguish frequent from unfrequent methods. The threshold is the sample size $size(c, e)$, where c is a confidence level e e is a margin of error. If the number of collected traces of a method is above the required sample size, it is said to be **frequent**.
Shareability. The method shareability gives how much the results of a method call are shared among different users, because if results of a method are shared among many users, caching this method may potentially increase the hit ratio. A method shareability $shareability(m)$ is the percentage of different user sessions in which requests lead to a method call with the same parameter values. Similarly to frequency, a method is said **shared** if its shareability is k standard deviations σ_{sh} above the shareability mean μ_{sh}, that is, $shareability(m) > \mu_{sh} + k \times \sigma_{sh}$. Anonymous method calls (not associated with any user) are not taken into consideration in this criterion.
Expensiveness. A method expensiveness is associated with the cost $cost(m)$ for computing it, which can be the time taken to compute it or consumed memory, for example. As above, a method is said **expensive** if $cost(m) > \mu_{ct} + k \times \sigma_{ct}$, where μ_{ct} and σ_{ct} are the cost mean and standard deviation, respectively.

needs to have its decision criteria analyzed in an objective way. However, they consists of *subjective* definitions. This means that, when adopting our pattern, developers must providing a meaning for each criterion. We thus, as part of our framework, propose objective measurements to evaluate each criterion. There are five specified measurements, which are detailed in Table 5.1—two of the criteria, associated with data size and cache size, are taken into account in the next activity.

The discussed analysis requires collected data. However, wrong conclusions can be reached due to a limited amount of data, such as considering a method with only a few executions as static, frequent, or less changing. In situations in which we have an insufficient amount of data, we assume *Not sure* as the answer of the decisions in the flowchart.

Based on our objective criteria evaluation and the flowchart of the Cacheability Pattern, our approach identifies cacheable methods. It is important to note that the analysis of a method may result in many cacheable opportunities (and consequently many entries in the cache), because our approach distinguishes and analyzes method calls, which are specified as a combination of the method signature and parameter values. In the next activity, described as follows, we detail how we choose those to be cached, using the two remaining criteria, not taken into account in this activity.

5.2.3 Cache Management: Caching Identified Opportunities

With the previous activities, we identify a set of cacheable methods. We now discuss how our approach manages the cache component to cache the selected method calls, as well as keep consistency. Similarly to the data tracking activity, through monitoring the application execution, we intercept calls to these cacheable method, and assess whether the content associated with the call is in the cache. If the cache had unlimited size, any content could be put in the cache. However, because this is often unrealistic, adding any content to size-limited caches can result in a problem of cache thrashing, where the cache is rapidly filled with data, and purges it soon after, without having a benefit of caching.

To make our approach less dependent on the effectiveness of the selected cache replacement algorithm (e.g., least recently used), we only add cached content when there is enough space in the cache, considering the data size and cache size, the two remaining concepts of the Cacheability pattern. When a cacheable method is called and the returned content is not in the cache, we estimate how much space of the cache this method requires, and verify whether the cache has the corresponding free space to allocate such content. If there is no enough space in the case, the content is not cached. Free space is obtained in the cache, when its current content is evicted, according to a selected eviction policy.

Our approach solely learns whether method executions should be cached, i.e. eviction is out of the scope of this work. A possible simple solution, which is that adopted by default in our approach, is to relax freshness and admit potential staleness of method calls to increase performance and scalability, by means of a time-to-live (TTL) eviction. With TTL, cached methods expire after a time in cache, regardless of possible changes. This would free cache space for caching new content associated with cacheable methods.

5.3 Framework

The activities of our approach, conceptually described above, were implemented as a framework that can be instantiated in web applications. Our framework is implemented in Java, thus can be used with Java web applications. This choice is due to our previous programming experience and tools available that were adopted as part of our implementation.

To collect data to be analyzed and manage cacheable methods, we intercept method executions using aspect-oriented programming (AOP), more specifically the AspectJ[1] implementation. AOP provides an easy way to weave code at specified points, without changing the base code. To address the challenge of recording large input and output objects, we summarize data as hash values, as described, using the MurmurHash3[2] hash function, and store application traces in a database. Hashing and saving actions are performed in an execution thread separate from the one that is processing the request, to minimize response delays. Our framework also provides means for developers to make customizations using hints, indicating possible locations of cacheable methods, which can improve the set of caching candidates as well as exclude methods that should not be cached, thus saving the time of tracking them.

Available solutions of caching components were also adopted. They provide APIs to manipulate data and access metadata information about the cache, such as statistics and configurations. Our framework is decoupled of particular caching components, and thus supports the most popular distributed cache systems and libraries, which can be configured through property files and annotations. Our framework automates the decision of what to cache, but such caching components allow us to collect metadata about the cache and also manipulate cached content easily.

The collected data are analyzed offline, separately from the web application, to prevent impact in the application performance—it can even run on a dedicated machine. To evaluate shared execution traces (accessed by multiple users), we obtain the user session, which is an application-specific information thus taken into account only in application-level caching. Our framework provides a set of alternative implementations to obtain this information from the most popular web frameworks, such as Java EE and the Spring Framework[3]. In case alternative ways of managing user sessions are adopted, developers should implement interfaces provided by our framework. This described implementation not only can be used to incorporate our proposal to web applications but was also used in our evaluation, which is presented next.

[1]<https://eclipse.org/aspectj/>
[2]<https://github.com/google/guava/wiki/HashingExplained>
[3]<https://spring.io/>

6 EVALUATION

In this chapter, we proceed to the evaluation of our automated caching approach. We first describe our evaluation procedure, and then discuss obtained results and threats to validity.

6.1 Procedure

To evaluate our approach, we used performance test suites, which aim to simulate real-world workloads and access patterns (BINDER, 2000). Furthermore, such tests have been used to evaluate improvements of caching in web applications (DELLA TOFFOLA; PRADEL; GROSS, 2015; CHEN et al., 2016). Simulations were performed using three different caching configurations: (i) no application-level caching (**NO**), (ii) application-level caching manually designed and implemented by developers (**DEV**); and (iii) our approach (**AP**). To assess performance, we used two metrics: *throughput* (number of requests handled per second); and *hit ratio*, given that they are well-known in the context of web applications and cache performance tests.

Our simulation emulated client sessions to exercise applications and evaluate decision criteria. Simulations consisted of 50 simultaneous users constantly navigating through the application, at a limit of 1000 requests per user. Each emulated client navigates from an application page to another, randomly selecting the next page from those accessible from the current page. The navigation process starts on the application home page. Emulated clients thus follow the same navigation rules of a real user. The evaluation of different criteria requires different parameters. We adopted 99% and 3% as the confidence level and margin of error, respectively, for the frequency criterion. For shareability and expensiveness, we adopted $k = 1$, while for changeability, $k = 0$. For expensiveness, the cost corresponds to the method execution time. Finally, we used information collected over 2 minutes to build the caching decision model.

Our evaluation was performed with three open-source web applications[1], presented in Table 6.1, which summarizes the general characteristics of each target system. To prevent application bias in our results, we selected applications with different sizes (6.3–111.3 KLOC) and domains. Cloud Store, in particular, is developed mainly for per-

[1]Available at <http://www.cloudscale-project.eu/>, <https://github.com/SpringSource/spring-petclinic/> and <http://www.shopizer.com/>.

Table 6.1: Target systems of our study.

Project	Domain	LOC	# Files	Database Properties
Cloud Store	e-Commerce based on TPC-W benchmark	7.6K	98	300K customer data and 10K items
Pet Clinic	Sample application	6.3K	72	6K vets, 10K owners and 13K pets
Shopizer	e-Commerce	111.3K	946	300K customer data and 10K items

formance testing and benchmarking, and follows the TPC-W[2] performance benchmark standard. It is important to highlight that the DEV configuration was implemented by developers independently from this work and, therefore, the guidelines and patterns presented in Chapter 4 were not considered in this implementation.

For Pet Clinic and Cloud Store, we used test cases written by their developers and developed test cases for Shopizer in which they are unavailable. For the latter, we created test cases to cover searching, browsing, adding items to shopping carts, checking out, and editing products.

We used Tomcat[3] as our web server and MySQL[4] as the DBMS. We used two machines located within the same network, one machine for the DBMS and web server (8G RAM, Intel i5 3.2GHz), and one machine for JMeter[5] (8G RAM, Intel i7 2GHz). We used EhCache as our underlying caching framework, because it is used for all target applications. Moreover, we used the same cache component configurations (replacement policy and in-memory size) as specified by developers in each application.

6.2 Results

We now proceed to the results of the evaluation of our automated caching approach, which was based on measuring three aspects: (i) performance; (ii) impact of monitoring overhead; and (iii) differences between developers' caching decisions and our approach.

6.2.1 Performance Evaluation

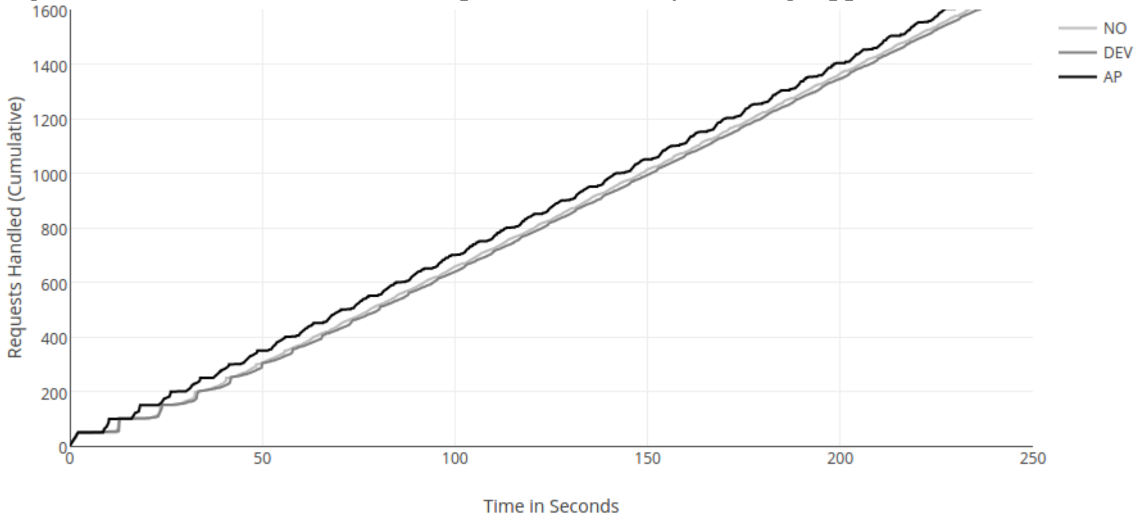
Based on our simulation, we observed that our approach (**AP**) improves the throughput of all target applications, in comparison with no use of caching (**NO**). Moreover,

[2]<http://www.tpc.org/tpcw/>
[3]<http://tomcat.apache.org/>
[4]<https://www.mysql.com/>
[5]<http://jmeter.apache.org/>

Table 6.2: Simulation Results: Throughput and Hit Ratio.

Application	NO	DEV		AP	
	Throughput	Throughput	Hit Ratio	Throughput	Hit Ratio
Cloud Store	7.02	7.03 (+0.14%)	0.66	7.03 (+0.14%)	0.97 (+31.9%)
Pet Clinic	30.83	33.02 (+6.63%)	0.99	35.10 (+12.16%)	0.99 (0%)
Shopizer	6.53	6.52 (-0.15%)	0.99	6.54 (+0.15%)	0.67 (-32.3%)

Figure 6.1: Cumulative Handled Requests *vs.* Time by Caching Approach for CloudStore.



when compared with the application-level caching manually implemented by developers (**DEV**), our approach achieves at least similar performance. Even if results were not as good as those obtained with DEV, they could be considered good because our approach automates an error-prone and time-consuming task performed by developers. Table 6.2 shows the throughput and hit ratio obtained for each individual target application.

Figures 6.1, 6.2 and 6.3 further shows the cumulative throughput over simulation time for each application. As can be seen, for the three studied applications, the throughput is similar in the beginning of the simulation, but DEV and AP only improve results with respect to NO after there is cached content. As soon as more requests are made, the benefit of caching can be gradually seen. However, only a few methods are usually cached through application-level techniques. Thus, the advantage of caching usually becomes apparent over an extended period of application execution. It is possible to observe a trend that the longer the test runs, the more benefit our approach provides, because it changes the set of cacheable methods with a continuous evaluation.

With respect to hit ratio, DEV is used as baseline for AP. Our approach shows a hit ratio improvement of 31.9% for the Cloud Store. Such improvement is related to caching *search* operations, because developers simply cache all search combinations, and our approach caches only those searches that can potentially improve application performance

Figure 6.2: Cumulative Handled Requests *vs.* Time by Caching Approach for Pet Clinic.

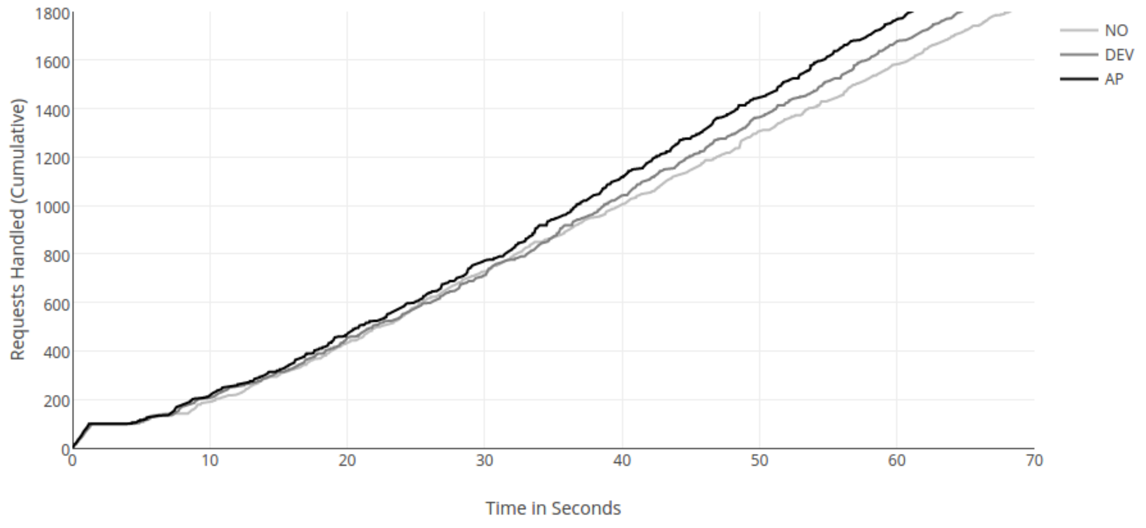
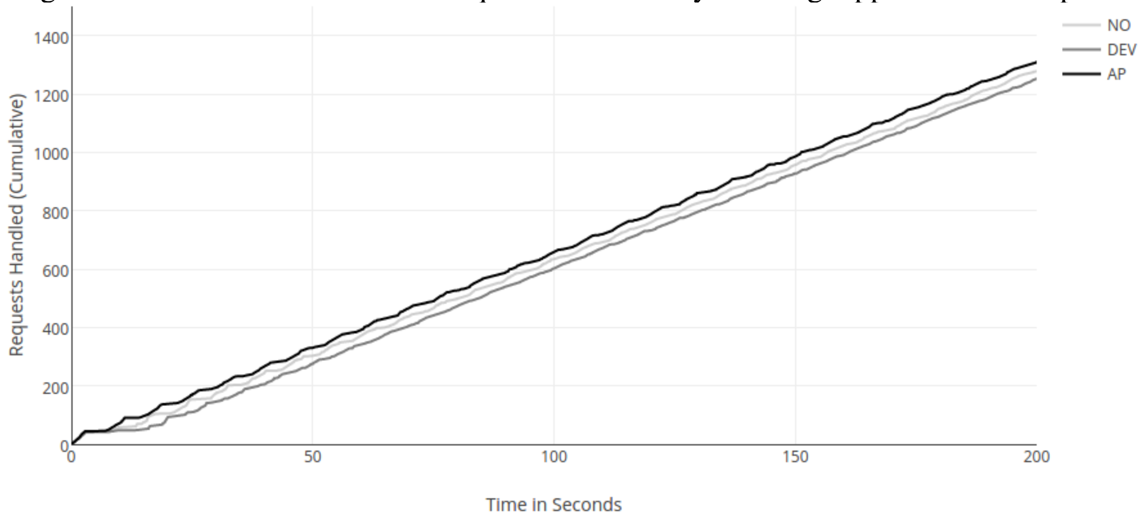


Figure 6.3: Cumulative Handled Requests *vs.* Time by Caching Approach for Shopizer.



(according to our evaluation criteria). This reduces the amount of cached data, freeing space in the cache, avoiding evictions and leading to higher hit ratios. For the Pet Clinic, AP achieves the same high hit ratio as DEV. The small performance improvement for Shopizer may be also related to the amount of cached data, which is higher than the cache size originally configured by developers. Consequently, AP cannot put all the cacheable methods in the cache, which results in the recomputation of cache-expected data, leading to lower hit ratio and, consequently, to a lower throughput. We observed a larger improvement provided by AP with the Pet Clinic application. After a manual investigation, we concluded that our approach caches much more method calls than developers. Consequently, caching them significantly reduced the network transfer time, and thus resulted in a large performance improvement.

Table 6.3: Simulation Results: Throughput of Each Configuration of Our Approach.

Application	NO	DEV		APLM		APMC		AP	
Cloud Store	7.02	7.03	(+0.14%)	7.00	(-0.28%)	7.03	(+0.14%)	7.03	(+0.14%)
Pet Clinic	30.83	33.02	(+6.63%)	25.58	(-17.02%)	29.51	(-4.28%)	35.10	(+12.16%)
Shopizer	6.53	6.52	(-0.15%)	6.33	(-3.06%)	6.44	(-1.37%)	6.54	(+0.15%)

As conclusion, our approach was able to discover cacheable method calls at runtime, based on the application workload, with an improvement of our baseline. Therefore, it can relieve the burden from developers of identifying and implementing caching. Furthermore, if there is no enough trust to allow our approach to managing the cache in a production environment automatically, its caching decisions can be used as a guideline to developers while developing application-level caching, which is not trivial mainly in large applications.

Our approach takes about 1 minute to analyze 2 million application traces, in an Intel i5 3.2GHz CPU with 8G RAM. It is only required to derive the set of cacheable methods. Such process is executed in background and thus has minimal impact on the application. Note that it can also be configured to run on a separate machine. However, we are aware that the expected overhead of monitoring all method calls can potentially minimize the performance improvement of our approach. Next, we proceed to the evaluation of such overhead.

6.2.2 Evaluation of the Monitoring Overhead

Although our approach can discover and cache cacheable method calls according to the current workload of an application, the expected overhead of tracing method calls is sometimes higher than the benefit of caching, which leads our approach to lower performance. To better evaluate the burden of monitoring and benefit of the cacheable opportunities, we executed each possible setup of our approach separately. Complementing the results presented in Table 6.2, two new scenarios were evaluated: (i) monitoring application execution without the benefit of caching (**APLM**), and (ii) monitoring and caching the identified cacheable opportunities (**APMC**). Table 6.3 shows the throughput of the applications of each running cycle of our approach. Again, we use NO as baseline and compare the throughput with DEV, APLM, APMC and AP caching.

As can be seen in Table 6.3, APLM decreases the application throughput in 0.28%, 17.02% and 3.06% for Cloud Store, Pet Clinic and Shopizer, respectively, when compared to NO. However, after the analyzes of the traces and identification of cacheable methods, the overhead of the monitoring cycle is reduced because the cache is now being used. It shows that our approach can potentially improve the application performance and automate this non-trivial task of selecting cacheable content, according to a specific workload.

Finally, we disabled the monitoring and let only the caching decisions of our approach running (AP), and then the throughput improvement is perceived in all of the applications. Even with such overhead, the performance of Cloud Store and Shopizer with APMC remains slightly the same as DEV. However, the cost of monitoring the application execution is sometimes higher than the benefit of caching. This is the case of AP for Pet Clinic, which is possibly because the application is small and few methods are cached, which does not counterbalance the cost of monitoring.

Although our approach implies an overhead to automatically identify and cache data, as mentioned in Chapter 4, while designing and implementing a caching solution, developers should rely on performance measurements to guide the design of an application-level caching solution. Furthermore, they should document and report such measurements in order to compare and reproduce performance tests employed. Consequently, the overhead of monitoring and analyzing data to make caching decisions also affects developers because this information is required either if such information is manually analyzed by developers, or automatically by an algorithm.

However, the analysis, design and maintenance effort required from developers before making substantial caching changes (that in our approach is automatically done) were not evaluated. Moreover, we do not investigate any technique to reduce the impact of collecting and storing data, such as collecting samples of requests. Both issues can be better explored in future work.

6.2.3 Comparison with Human-made Caching Decisions

For each application, there may exist many possible cacheable method calls. Thus, we also compared the number of caching opportunities that were selected and managed by our approach with the choice made by developers, implemented in the target applications. Table 6.4 shows that our approach not only caches those methods selected by developers, but many others. However, the number of selected methods is small in comparison with all possible methods, as shown in the column AP.

Results indicate that developers may be conservative while identifying cacheable methods, and select only those methods in which there is a strong confidence that caching

Table 6.4: Number of Cacheable Methods: Our Approach *vs.* Human-made Decisions.

Application	**Total**	**DEV**	**AP**	**Intersection**
Cloud Store	812	4	11	4
Pet Clinic	205	1	4	1
Shopizer	5212	15	65	15

them increases the hit ratio. For instance, in Pet Clinic the number of vets is often the same. Such opportunities are always detected by our approach. However, our approach was able to identify more cacheable opportunities, which justifies the performance improvement. Moreover, while implementing a cache solution, developers usually select *cacheable methods*, and thus any call to a cacheable method is cached. This could lead to a higher and ineffective use of the cache size, because not all method calls are frequent or expensive. Our approach can deal with specific method calls, leading to an optimal utilization of the cache infrastructure.

As conclusion, our approach identifies a higher number of cacheable methods to a fraction (63–76%), when compared to the total number of cached methods identified by developers. For large applications like Shopizer, manually analyzing and identifying 65 possible cacheble methods may be time-consuming or not even be possible.

6.3 Threats to Validity

The performance benefits of caching highly depends on workloads. It is possible that the adopted workload from the performance tests may not be representative enough, given that we do not make any assumption regarding the workload when conducting our experiments, and rely on the randomness added to the tests.

Nevertheless, our approach does not depend on a particular workload and can find cacheable methods with any pre-specified workload, which may evolve over time in real world scenarios. Therefore, even if the workload changes substantially and initial cacheable methods are no longer useful, our approach can adapt itself, automatically discard old caching configurations and discover a new set of cacheable methods.

Our evaluation involves only three target applications, therefore results may not be generalizable. However, to address this threat, we selected open-source applications, with different sizes and domains.

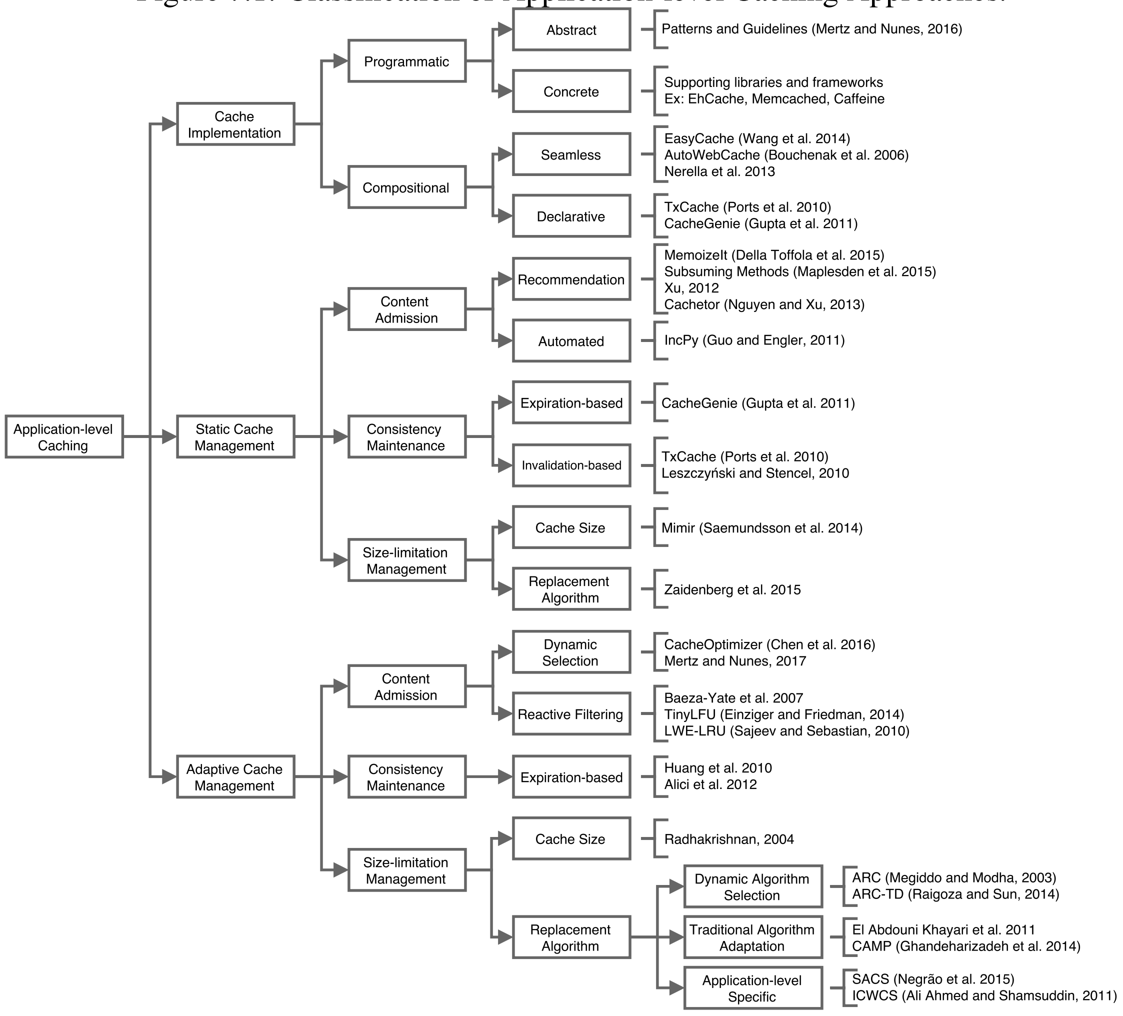
7 RELATED WORK

We performed a comprehensive survey of application-level caching in web applications (MERTZ; NUNES, 2017), which allowed us to derive a taxonomy to classify research that was done in this context. This taxonomy is presented in Figure 7.1, which takes into account the issues and challenges that were introduced when we discussed web caching and application-level caching, in previous sections. In this chapter, we focus specifically on research work that addresses admission issues given that our proposed approach is focused on the identification of cacheable content.

It is important to mention that our initial intention was to perform a systematic mapping of application-level caching approaches. However, many alternative investigated search strings led to either a small set of papers, with many related works that were not retrieved, or a large set that included many unrelated works. Such large set was unfeasible to be processed in a timely fashion. The first issue we faced was the ambiguous terminology associated with the context of our research. Given the different types of caching that can be employed through the web infrastructure and the lack of specific nomenclature that identifies each caching, it is difficult to match solely papers that deal with application-level issues. Furthermore, terms such as *adaptive*, *web application*, and *cache*, associated with our study, are widely used in many other contexts. As result, proceeding with a systematic approach would result in a poor review of application-level caching. Therefore, we conducted our survey with those studies we collected using a less rigorous selection process. We collected many relevant papers using alternative query strings, and further searched for papers published by key authors or cited in relevant papers, using a snowball approach. This gives us confidence that relevant papers are indeed included in our survey.

Application-level caching approaches are classified into three categories, depending on how the approach helps developers with caching issues. We next discuss these approaches, following such categorization. First, we present work focused on supporting the implementation of a caching solution, which is one of our challenges. Then, static approaches regarding content admission are presented and discussed. Finally, based on challenges of application-level caching and its adaptation requirements, we finally introduce caching approaches focused on content admission that are able to adapt its behavior, typically by means of a feedback loop.

Figure 7.1: Classification of Application-level Caching Approaches.


Application-level Caching
Cache Implementation
Programmatic
Abstract
Patterns and Guidelines (Mertz and Nunes, 2016)
Concrete
Supporting libraries and frameworks
Ex: EhCache, Memcached, Caffeine
Compositional
Seamless
EasyCache (Wang et al. 2014)
AutoWebCache (Bouchenak et al. 2006)
Nerella et al. 2013
Declarative
TxCache (Ports et al. 2010)
CacheGenie (Gupta et al. 2011)
Static Cache Management
Content Admission
Recommendation
MemoizeIt (Della Toffola et al. 2015)
Subsuming Methods (Maplesden et al. 2015)
Xu, 2012
Cachetor (Nguyen and Xu, 2013)
Automated
IncPy (Guo and Engler, 2011)
Consistency Maintenance
Expiration-based
CacheGenie (Gupta et al. 2011)
Invalidation-based
TxCache (Ports et al. 2010)
Leszczyński and Stencel, 2010
Size-limitation Management
Cache Size
Mimir (Saemundsson et al. 2014)
Replacement Algorithm
Zaidenberg et al. 2015
Adaptive Cache Management
Content Admission
Dynamic Selection
CacheOptimizer (Chen et al. 2016)
Mertz and Nunes, 2017
Reactive Filtering
Baeza-Yate et al. 2007
TinyLFU (Einziger and Friedman, 2014)
LWE-LRU (Sajeev and Sebastian, 2010)
Consistency Maintenance
Expiration-based
Huang et al. 2010
Alici et al. 2012
Size-limitation Management
Cache Size
Radhakrishnan, 2004
Replacement Algorithm
Dynamic Algorithm Selection
ARC (Megiddo and Modha, 2003)
ARC-TD (Raigoza and Sun, 2014)
Traditional Algorithm Adaptation
El Abdouni Khayari et al. 2011
CAMP (Ghandeharizadeh et al. 2014)
Application-level Specific
SACS (Negrão et al. 2015)
ICWCS (Ali Ahmed and Shamsuddin, 2011)

7.1 Cache Implementation

In order to reduce the effort demanded by developers while implementing application-level caching, implementation-centered approaches have been extensively developed. Such approaches focus on the provision of solutions that raise the abstraction level of caching and reduce a significant amount of cache-related code to be added to the base code. Therefore, a research challenge in application-level caching is how to *ease the implementation* of a caching system for developers.

Usually, the idea behind solutions to implementation issues is that implementation effort can be reduced by providing a system or library that raises the level of abstraction and handles some caching operations, freeing developers to write the most relevant code (i.e. business logic). As shown in Figure 7.1, approaches related to caching implementation can be categorized into two types, concerning the adopted programming model:

programmatic and *compositional*. The former approach is a per-application solution and requires code changes to take advantage of the caching options. For example, EhCache provides a full-featured caching component, which is responsible for dealing with memory and disk store. However, it requires coding with the EhCache provided classes or annotations in order to perform the caching operations. The latter has a lower impact in the application, as it does not require to introduce code interleaved with its base code.

7.1.1 Programmatic

The simplest programmatic solution for application-level caching consists of the use of *abstract* solutions such as design and implementation patterns. The results of our qualitative research, presented in Chapter 4, fit in this category because it provides a set of patterns and guidelines to support developers while designing, implementing and maintaining application-level caching. Although simple, such solutions may require a significant amount of changes in the base code to be adopted, and may not provide an adequate degree of transparency and flexibility to developers because they are abstract solutions and should be manually implemented.

In order to provide *concrete* support, libraries and frameworks have been developed, providing useful and ready-to-use cache-related features. Examples of such solutions are distributed cache systems, e.g. Redis[1] and Memcached, and libraries that cache content locally, e.g. Spring Caching[2], EhCache, Infinispan[3], Caffeine[4] and Rails low-level caching[5]. Although these advantages, these concrete solutions still leave the responsibility of managing the cache entirely to developers, which can result in increased complexity and additional bugs. Furthermore, adopting such solutions not only involves a learning curve and significant programming effort from developers to maintain data in the cache, but also couples a caching system to the web application, which adds complexity to the application and reduces the possibility of reuse within it (PORTS et al., 2010; WANG et al., 2014; ZHANG; LUO; ZHANG, 2015). For a deeper analysis of representative cache systems/libraries and their main techniques regarding data management, we refer the reader to elsewhere (ZHANG; LUO; ZHANG, 2015).

[1]<https://redis.io/>
[2]<https://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html>
[3]<http://infinispan.org/>
[4]<https://github.com/ben-manes/caffeine>
[5]<http://edgeguides.rubyonrails.org/caching_with_rails.html#low-level-caching>

7.1.2 Compositional

The drawback of concrete solutions is partially addressed by *compositional* approaches, which can relieve part of the developers' burden by automating tasks through *declarative* or *seamless* approaches. The former is based on the provision of knowledge associated with the semantics of application code and data. In this case, annotations referred to as *assertions* or *contracts*, are used to describe application properties, which can be processed at runtime to execute specific tasks. Such annotations have become widely adopted in dynamic programming languages (STULOVA; MORALES; HERMENEGILDO, 2015).

Therefore, they can potentially maximize the performance improvement without comprising other requirements. *CacheGenie* (GUPTA; ZELDOVICH; MADDEN, 2011) is a system that provides high-level caching abstractions for frequently observed query patterns in web applications. These abstractions take the form of declarative query objects and, once developers specify them, insertions, deletions, and invalidations are done automatically. CacheGenie is based on triggers inside the database to automatically invalidate the cache, or keep it synchronized with the data source, as expressed by the developer in the application code. Similarly, Ports et al. (2010) also offer a simple programming model, namely *TxCache*, for developers. They simply designate certain functions as cacheable, and TxCache automatically caches those marked method results. CacheGenie and TxCache also provide automatic consistency management, which is better explored in Section 7.2.

Seamless solutions are those that are coupled to the application in a transparent way, being added to the application, for example, as a surrounding layer, such as database and web proxy, without the need for refactoring application code. Such solutions are especially useful in scenarios where the application was not conceived to use caching since the beginning, and requests for performance and scalability improvements emerged after many releases. Such approaches come as an alternative to refactoring the application with the introduction of programmatic approaches. Although faster results may be obtained with transparent solutions, they can be hard to support and tune, because system administrators, or even developers, might need to be specially trained or experienced in particular caching solutions and scenarios to configure them properly.

While the majority of seamless approaches are focused on database (RAVI; YU; SHI, 2009) or proxy-level caching (ALI; SHAMSUDDIN; ISMAIL, 2011), there are some studies focused on application-level. *EasyCache* (WANG et al., 2014) provides transparent cache pre-loading, access and consistency maintenance without extensive modifications to the application or a complete redesign of the database. Such approach combines properties of mid-tier database caching and application-level caching to build a mechanism that loads data from an existing database into the memory of the application server, and translates database queries into application-level objects. Therefore, Easy-Cache transfers the load from the database to a distributed cache. However, in order to keep the approach less expensive in terms of memory and network bandwidth, complex queries (e.g. nested and statistics queries) are unsupported, as well as large objects.

In order to ease the caching implementation, aspect-oriented programming (AOP) (KICZALES et al., 1997) has been explored by increasing modularity with an improved separation of cross-cutting concerns. Bouchenak et al. (2006) demonstrated that dynamic web caching can be considered a cross-cutting concern and, therefore, AOP methods are used as a flexible and easy-to-use tool to develop the middleware support. They proposed a caching middleware named *AutoWebCache*, which caches dynamic web pages at the front-end when a cache miss occurs—i.e. they use a reactive approach—while maintaining consistency with the back-end databases through effective cache invalidation policies. The AutoWebCache prototype uses AOP to add caching of dynamic web pages to a servlet-based web application that interfaces a database with JDBC. This methodology is general enough to encompass other sources of dynamic data. Specifically, individual aspects can be developed separately for each source and then woven together. Also based on AOP, Nerella et al. (2013) described an approach to cache the results of join sub-queries, with the goal of retrieving partially query results from the cache, rather than whole queries. The approach relies on histograms built from the data collected at runtime to estimate the cacheability of joins and predicates to construct query plans and determine which join sub-queries to cache. Information from previous executions of the same query during runtime is used during the construction of the query plans, even when the data has changed between these executions. Furthermore, the cache is maintained incrementally, when the underlying collections change and the use of the cache space is optimized by a cache replacement policy.

Applications can also be developed with business logic at the client-side, typically with the support of JavaScript frameworks. At the client-side, cache strategies are usually predefined by web service providers and brought into effect by browsers. Nevertheless, popular JavaScript frameworks merely cache the data from the entire web pages or request responses, providing inadequate flexibility for developers. Huang et al. (2010) proposed a browser-side caching framework for web-delivered services composition, which provides to developers an easy way to choose among pre-defined strategies. The framework does not require developers to manage the cache content manually but allows developers to customize their caching strategies, such as setting expiration time, cache granularity, and replacement policies.

Although programmatic and compositional implementation approaches can raise the abstraction level of caching and prevent adding much cache-related code to the base code, their limitation is that they still require design reasoning, such as deciding whether to cache content. Therefore, a more fine-grained solution would require minimal effort and input from developers, which are the focus of the approaches described next.

7.2 Static Content Admission

Static application-level approaches are those that address design and maintenance issues of caching focusing on automating required tasks or giving suggestions towards easing the developer reasoning. According to the taxonomy presented in Figure 7.1, static cache management solutions are classified into three categories, depending on which caching issue they address. As previously mentioned, we only present approaches focused on content admission.

Application-level caches allow caching at a granularity adequate to the application, providing a way to cache entire HTML pages, page fragments, database queries or even computed results. Since arbitrary content can be cached, opportunities for caching emerge in the most diverse parts of the application. However, estimating content cacheability to select those that would improve the application if cached is not trivial, mainly in complex applications. If the developer fails to get this right, it can end up reducing application performance instead of improving it, by consuming more cache memory and at the same time suffering from cache misses, where the data is not getting served from the cache, but fetched from the source. There are two main ways of helping developers select cacheable content: providing caching *recommendations*, or providing an *automated* admission by automatically identifying and caching content. Table 7.1 summarizes the caching approaches that deal with admission issues.

The first way of helping developers while admitting content to the cache is by recommending improvement opportunities. Approaches in this context are usually based on

Table 7.1: Static Caching Approaches for Selection of Cacheable Content.

	Approach	Based on	Content	Data Input	Analysis	Output
Recommendation	MemoizeIt (DELLA TOFFOLA; PRADEL; GROSS, 2015)	Iterative profiling	Method calls	Time, frequency, and input-output profiling	Hit ratio, invalidation and size thresholds, and estimations	Report with ranked list of potential opportunities to the user for manual inspection
	Subsuming Methods (MAPLESDEN et al., 2015)	Application profiling	Method calls	Calling context tree	Customized metric based on method distance in the context tree	Subset of the methods in an application that are interesting from a performance perspective
	Xu (2012)	Application profiling	Data Structures	Heap data structures for each allocation site during the execution of the application	Customized metric to approximate reusability based on three different levels (instance, shape and data)	Report with a list of top potentially reusable allocation sites to the user for manual inspection
	Cachetor (NGUYEN; XU, 2013)	Abstract dependency graph analysis	Byte-code instructions, data structures and method calls	Instructions, data structures and call sites profiling	Cacheability measurements for each type of content based on frequency	Ranked list of potential opportunities
Automated	IncPy (GUO; ENGLER, 2011)	Application profiling	Method calls	File accesses, value accesses, and method calls	Safeness (consistency) and worthwhileness (expensiveness) heuristics	Automatically caches and invalidates data

the analysis of application profiling information, which can capture application-specific details through monitoring its execution when facing different situations. Traditional profiling approaches typically record measurements of method calls. Usually, cost measurements are captured with calling context information, which conveys the hierarchy of active methods calls of a request.

Della Toffola, Pradel and Gross (2015) addressed this problem by identifying and suggesting method-caching opportunities. Their approach, called *MemoizeIt*, is based on comparing inputs and outputs of method calls and dynamically identifying redundant operations, which may be avoided by reusing already computed results for particular inputs. To prevent the overhead of comparing objects for all method invocations in detail, MemoizeIt first compares objects without following any object references, and then iteratively increases the depth of exploration while shrinking the set of considered methods. By doing this, after each iteration, the approach ignores methods that cannot benefit from memoization, allowing it to analyze calls to the remaining methods in more detail. Also by analyzing method calls in an application profile, the work of Maplesden et al. (2015) can identify the entry point to repeated patterns of method calls, which are called *subsuming methods* and are identified by analyzing the smallest parent distance among all the common parents of a method.

Xu (2012) focused on a common problem in object-oriented applications, the frequent creation of data structures (by the same allocation site), whose lifetimes are disjoint, and shapes and data content are always the same. He follows the same principle of helping developers and report a list of top allocation sites that create such data structures. Then developers can manually inspect the code and implement the appropriate solution to improve performance. *Cachetor*, proposed by Nguyen and Xu (2013), addresses bloats in the work repeatedly done to compute the same data values by suggesting spots of invariant data values that could be cached for later use. They proposed a runtime profiling tool, which uses a combination of dynamic dependency and value profiling to identify and report operations that keep generating identical data values. However, Cachetor makes strong assumptions about the programming language, such as the presence of a specialized type system, which are not held in dynamically-typed languages. Infante (2014) addressed the type system restriction by proposing a multi-stage profiling technique that uses dynamically collected data to reduce the profiling overhead.

Although these approaches can reduce complexity and time required by caching design, developers should still review the recommendations and decide whether to cache or not the suggested opportunities. Besides this selection process, developers still need to refactor the code manually, integrating cache logic into the application. Thus, a second way of helping developers while dealing with content admission is to automatically identify and cache the cacheable content, as opposed to just report potentially cacheable methods. However, such approaches require not only ways to analyze the application behavior, but mechanisms to manage cache and application at runtime.

Guo and Engler (2011) achieved this by implementing a technique, namely *IncPy*, as a custom open-source Python interpreter. IncPy explores the repetitive creation and processing of intermediate data files, which should be properly managed by developers to multiple dependencies between their code and data files, otherwise their analyses produce wrong results. To enable developers to iterate quickly without needing to manage intermediate data files, they added a set of dynamic analyses to the programming language interpreter so that it automatically caches the results of long-running pure method calls to disk, manages dependencies between code and on-disk data, and later re-uses results, rather than re-executing those methods, when it is guaranteed that it is safe to do so. Furthermore, this approach also allows developers to customize the execution by inserting annotations, which can force IncPy to always or never cache particular methods.

A key limitation of the approaches presented in this section is that, due to the non-adaptive nature of these solutions, they do not automatically consider changes in application workload and access patterns, which can lead such approaches to a poor or sub-optimal performance before the revision of caching decisions. Furthermore, such

Table 7.2: Comparison of Adaptive Application-level Caching Approaches for Admission.

Reference	**Monitored Data**	**Analysis**	**Behavior**	**Operation**	**Decision**	**Goal**
CacheOptimizer (CHEN et al., 2016)	Web server and database access logs	Static source code analysis and dynamic weblog analysis with colored Petri nets	Proactive	Offline	Content miss ratio threshold	Dynamic identification of cacheable content and definition of caching configurations
Baeza-Yate et al. (2007)	Query metadata and historic usage information	Utility-function based	Reactive	Online	Utility-function based on the probability of a query generate future cache hits	Dynamic decision whether it is worth to admit queries to the cache
TinyLFU (EINZIGER; FRIEDMAN, 2014)	Cache size (number of items it can store) and historic usage information	Frequency histogram	Reactive	Online	Utility-function based on the potential hit ratio increasing	Dynamic decision whether it is worth to admit content when an eviction is required
LWE-LRU (SAJEEV; SEBASTIAN, 2010)	Content attributes and traffic parameters	Multinomial logistic regression model	Reactive	Offline and online phase	Utility-function based on content worthiness factor	Dynamic admission and eviction decisions

approaches usually do not take application specificities into account and require human intervention for configuring, customizing and tuning the solution to the application requirements and needs, which means that most of the reasoning, as well as the integration with the tool, should be accomplished by developers.

7.3 Adaptive Content Admission

We now focus on approaches that can dynamically evaluate the cacheability status towards discovering cacheable data, i.e. whether a certain content should be cached. As previously mentioned, admission approaches help developers while selecting caching opportunities. This task is particularly complex because selected opportunities must continuously be revised, due to the changing workload characteristics and access patterns, or even the application evolution. This shortcoming motivates the need for adaptive caching solutions, which can automatically improve themselves to cope with the changing dynamics of the application while minimizing the challenges it poses for application developers.

As shown in Figure 7.1, adaptive admission-focused approaches can be seen from two different perspectives, depending on the purpose of the adaptation: (a) *dynamic selection* of the best cacheable opportunities through the evaluation of cacheability properties, and (b) *reactive filtering* of content that was previously identified as cacheable but should not anymore, due to some particular reason. Table 7.2 summarizes the surveyed adaptive caching approaches that deal with admission issues.

The automatic identification of caching opportunities at the application level is addressed by *CacheOptimizer* (CHEN et al., 2016), which monitors readable weblogs to create mappings between workload and database access. The analysis part consists of two processes: a static code analysis to identify possible caching configuration spots, and a characterization with colored Petri nets, which models the transition of states of an application based on weblogs from the web server. It can reach a global optimal caching decision, instead of focusing on top cache accesses, using a greedy approach. Differently from other approaches, CacheOptimizer is not implemented as a new caching framework; instead, it is integrated with those existing, automating the caching configuration according to the results of its analysis. Although this approach addresses method caching opportunities, it focuses on database-centric web applications; thus, only database-related methods are cached. Our approach share commonalities with CacheOptimizer, however we focus on general application methods while searching for cacheable options.

Reactive filtering approaches act at runtime by dynamically evaluating the cacheability of content that was previously identified as cacheable, according to the workload and access pattern, in order to avoid putting unworthy content in the cache. Such approaches are specifically useful when dealing with search engines, which usually have good opportunities for caching. However, not all search combinations will be frequently requested; thus it is important to find those searches that can potentially improve application performance if cached. It also can reduce the amount of cached data, freeing space in the cache, avoiding evictions and leading to higher hit ratios.

Baeza-Yate et al. (2007) addressed reactive filtering in a different way, by identifying infrequent queries of web search engines, which cause a reduction in hit ratio because caching them often does not lead to hits. Therefore, this approach can prevent infrequent queries from taking space of more frequent queries in the cache. The proposed cache monitors query executions and has two fully-dynamic parts. The first part is an admission controlled cache that only admits those queries that the admission policy, which analyzes and classifies queries as future cache hits. All queries that the admission policy rejects are admitted to the second part of the cache, an uncontrolled cache. Both caches implement a regular cache policy, more specifically, LRU. The uncontrolled cache can, therefore, manage queries that are infrequent but appear in short bursts, considering that the admission policy will reject queries that it concludes to be infrequent. Thus, the uncontrolled cache can handle cases in which infrequent queries may be asked again by the same user or within a short period. It guarantees that fewer infrequent queries enter the controlled cache, which is expected to handle temporal locality better. The admission policy uses either stateless features, which depend exclusively on the query, or stateful features, based on historical usage information.

Also focusing on filtering content, Einziger and Friedman (2014) proposed *TinyLFU*, which uses a frequency-based admission policy to boost the effectiveness of caches subject to skewed access distributions. Although TinyLFU acts reactively only when the cache is full, rather than deciding which cached content to evict, it decides, based on the recent access history, whether it is worth admitting an accessed content into the cache at the expense of the eviction of a candidate. To achieve such behavior, TinyLFU uses an approximate LFU structure, which maintains a fair representation of the access frequency of recently accessed contents, and then it is possible to trade-off the cost of eviction and the usefulness of the new content. A small variation of such approach is currently available to developers as a default replacement policy of the Caffeine caching framework, due to its high hit rate and low memory footprint.

Admission approaches can also be implemented by using sophisticated learning techniques. Sajeev and Sebastian (2010) proposed a semi-intelligent admission technique using the multinomial logistic regression (MLR) as a classifier. The MLR model is trained with previously collected traces and sanitized logs for classifying the web cache's content worthiness. The parameter content worthiness is computed using six parameters, which depend on the traffic and the content properties. The MLR model has a single output for an content, which is its worthiness class. Then, at runtime, when there is an incoming content, its worthiness class is computed or updated. If the content is new (not cached before), then admission control mechanism is invoked. The admission control mechanism uses a threshold based on the ratio of loading time and size of the content to decide whether the content should be admitted.

7.4 Discussion

Differently from programmatic solutions to application-level caching implementation, such as popular caching libraries and frameworks, our proposed caching approach does not require any additional implementation, detaching caching concerns from the application. Furthermore, our framework can automatically capture all the needed application-specific information to achieve its objectives (i.e. select and cache cacheable content), as opposed to other solutions (PORTS et al., 2010; GUPTA; ZELDOVICH; MADDEN, 2011), which demand input and configuration from developers. Despite not being required, our framework also allows developers to provide additional knowledge by using a declarative approach, which can improve the solution.

Compared to other admission solutions, our approach differs from them in the sense that we address complex logic and personalized content, which are produced and handled by methods of web applications. Moreover, we consider all application methods as cacheable options and do not focus on specific methods or types of web applications, as some approaches do (GUO; ENGLER, 2011; MAPLESDEN et al., 2015; DELLA TOFFOLA; PRADEL; GROSS, 2015; CHEN et al., 2016).

Even though there are approaches that focus on providing application-level caching with an adaptive behavior, only a few take application specificities into account to autonomously manage their target. In addition to the traditionally explored cache-related access information and statistics (BAEZA-YATE et al., 2007; SAJEEV; SEBASTIAN, 2010; EINZIGER; FRIEDMAN, 2014), our approach considers caching metadata conveyed by the application during its execution, such as cost to retrieve, user sessions, cache size and data size. Such metadata are in fact information that developers use while designing and implementing application-level caching, and thus enrich the application model with valuable application-specific information regarding the applicability of caching.

8 CONCLUSION AND FUTURE WORK

Application-level caching has been increasingly used in the development of web applications, in order to improve their response time given that they are becoming more complex and dealing with larger amounts of data over time. Caching has been used in different locations, such as proxy servers, often as seamless components. Application-level caching allows caching additional content taking into account application specificities not captured by off-the-shelf components. However, there is limited guidance to design, implement and manage application-level caching, which is often implemented in an *ad-hoc* way.

This dissertation consists of a step toward the systematic development and automation of application-level caching by providing an understanding of its state-of-practice and fruitful insights into how cache-related tasks can be automated, thus providing developers with substantial support to design, implement and maintain application-level caching solutions. We not only provided such understanding by means of structured knowledge derived from a qualitative study (in the form of patterns and guidelines), and a survey of the state-of-the-art on static and adaptive caching approaches, but also proposed a technique, and associated implemented framework, that together automate the caching of web application methods, by monitoring system execution and adaptively managing caching decisions at runtime.

Our work provides insights into the theory of adaptive application-level caching solutions and brings important points to the attention of researchers and practitioners. Next, we detail contributions and discuss future work.

8.1 Contributions

Given the results presented in this dissertation, we list below our main contributions.

Survey of Application-level Caching in Web Applications. We presented (Chapter 2 and 7) a survey that presents the current state-of-the-art research on static and adaptive application-level caching approaches. Such survey can be used as a start point for researchers and developers, who aim to improve application-level caching or need guidance in designing more robust caching systems, with humans out-of-the-loop.

Although we only presented approaches related to content admission in Chapter 7, such survey includes other caching issues and is currently under revision (MERTZ; NUNES, 2017).

Qualitative Study. We described (Chapter 3) the design and results of a qualitative study (MERTZ; NUNES, 2016a) that provides an in-depth analysis of how developers employ application-level caching in web-based applications. The study consisted of the selection ten web applications and investigation of caching-related aspects, namely design, implementation and maintenance practices.

Guidelines and Patterns. As a result of our qualitative study, we provided (Chapter 4) patterns and guidelines for caching design, implementation and maintenance to be adopted under different circumstances while modeling an application-level caching component.

Automated Caching Approach. We proposed an approach (MERTZ; NUNES, 2016b) (Chapter 5), that is focused on integrating caching into web applications in a seamless and straightforward way, providing an automated and adaptive management of cacheable methods.

Implementation of the Proposed Caching Approach. One of the goals of our study is to demonstrate the practical automation of caching tasks. Therefore, we described (Chapter 6) an implementation of our seamless and automated approach as a framework that manages the cache according to observations made by monitoring a web application at runtime, as well as detaches caching concerns from the application. We evaluated our approach empirically with three open-source web applications, and the results indicate that we can identify adequate caching opportunities by improving application throughput up to 12.16%.

As result, our contributions can support developers by providing guidance and an automated approach to address issues related to the development of an application-level caching solution. As opposed to related work that addresses solely specific content, such as database-related methods or web page content, our contributions can be applied to cache results of any computation done by a web application, which includes complex logic and personalized web content.

Furthermore, our automated approach takes into account application-specific details in caching decisions, which in fact is the information considered by developers in the development of a cache solution. Thus, our approach can reduce the reasoning required from developers. In addition, the integration between framework and application is seamless and does not require manual inputs. Finally, the automatically selected cache configuration reflects the monitored application workload, thus being sensitive to the application changing dynamics.

8.2 Future Work

The contributions provided by this dissertation are a step towards the development of new solutions to support developers while designing, implementing and maintaining application-level caches. Our work still has limitations that lead to opportunities of future work. We discuss them next.

Improvement of our Approach. Although the processing phase of our approach seems to be fast enough to provide cacheable opportunities in a timely fashion, the overhead of the data tracking activity should be further evaluated regarding scalability. Techniques for reducing the amount of data required to identify cacheable methods should be investigated, towards reducing the overhead and providing a faster model building.

Automation of Other Cache-related Tasks. Future work also involves extending the approach to deal with other caching issues towards reducing the developers' effort when designing and implementing application-level caching. Guidelines and patterns derived from our qualitative study that were not explored in terms of automation in this work can be used as foundation to design and implement caching solutions for these issues.

In summary, the work presented in this dissertation advances work on understanding and automating application-level caching in three main directions: a comprehensive picture of the different approaches proposed to support application-level caching, guidelines and patterns for guiding its development, and automation of content admission, which is an important cache-related task. Clearly there is still much to do in order to minimize the challenges it poses for developers, providing a better experience with caching for developers and an optimal usage of the caching infrastructure. Our work consists of a significant step towards this.

REFERENCES

Ali Ahmed, W.; SHAMSUDDIN, S. M. Neuro-fuzzy system in partitioned client-side Web cache. **Expert Systems with Applications**, v. 38, n. 12, p. 14715–14725, nov 2011. Available from Internet: <http://dx.doi.org/10.1016/j.eswa.2011.05.009>.

ALI, W.; SHAMSUDDIN, S. M.; ISMAIL, A. S. A survey of Web caching and prefetching. **International Journal of Advances in Soft Computing and Its Applications**, International Center for Scientific Research and Studies, v. 3, n. 1, p. 18–44, mar 2011. Available from Internet: <http://eprints.utm.my/36316/2/IJASCA15_A-Survey-of-Web-Caching.pdf>.

AMZA, C.; SOUNDARARAJAN, G.; CECCHET, E. Transparent caching with strong consistency in dynamic content web sites. In: **Proceedings of the 19th annual international conference on Supercomputing**. New York, New York, USA: ACM Press, 2005. p. 264. Available from Internet: <http://dx.doi.org/10.1145/1088149.1088185>.

BAEZA-YATE, R. et al. Admission Policies for Caches of Search Engine Results. In: **Proceedings of the 14th international conference on String processing and information retrieval**. Santiago, Chile: Springer-Verlag, 2007. v. 4726, p. 74–85. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-75530-2_7>.

BALAMASH, A.; KRUNZ, M. An overview of web caching replacement algorithms. **IEEE Communications Surveys & Tutorials**, v. 6, n. 2, p. 44–56, 2004. Available from Internet: <http://dx.doi.org/10.1109/COMST.2004.5342239>.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. **Encyclopedia of Software Engineering**, v. 2, p. 528–532, 1994. Available from Internet: <http://dx.doi.org/10.1.1.104.8626>.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. **IEEE Transactions on Software Engineering**, IEEE, SE-12, n. 7, p. 733–743, jul 1986. Available from Internet: <http://dx.doi.org/10.1109/TSE.1986.6312975>.

BINDER, R. **Testing object-oriented systems: models, patterns, and tools**. [S.l.]: Addison-Wesley, 2000. 1191 p.

BORSTLER, J.; PAECH, B. The Role of Method Chains and Comments in Software Readability and Comprehension – An Experiment. **IEEE Transactions on Software Engineering**, IEEE, p. 1–1, 2016. Available from Internet: <http://dx.doi.org/10.1109/TSE.2016.2527791>.

BOUCHENAK, S. et al. Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. **Proceedings of the ACM/IFIP/USENIX International Conference on Middleware**, Springer Berlin Heidelberg, Berlin, Heidelberg, v. 4290, p. 1–21, nov 2006. Available from Internet: <http://dx.doi.org/10.1007/11925071>.

CANDAN, K. S. et al. Enabling dynamic content caching for database-driven web sites. **Proceedings of the ACM SIGMOD International Conference on Management of Data**, ACM, v. 30, n. 2, p. 532–543, jun 2001. Available from Internet: <http://dx.doi.org/10.1145/376284.375736>.

CARVAJAL, L. et al. Usability through software design. **IEEE Transactions on Software Engineering**, IEEE, v. 39, n. 11, p. 1582–1596, nov 2013. Available from Internet: <http://dx.doi.org/10.1109/TSE.2013.29>.

CHEN, T.-H. et al. CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-based Database-centric Web Applications. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016**. New York, New York, USA: ACM Press, 2016. p. 666–677. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950303>.

DELLA TOFFOLA, L.; PRADEL, M.; GROSS, T. R. Performance problems you can fix: a dynamic analysis of memoization opportunities. In: **Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015**. New York, New York, USA: ACM Press, 2015. p. 607–622. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2814270.2814290>.

DOMÈNECH, J. et al. Web prefetching performance metrics: A survey. **Performance Evaluation**, Elsevier Science Publishers B. V., v. 63, n. 9-10, p. 988–1004, oct 2006. Available from Internet: <http://dx.doi.org/10.1016/j.peva.2005.11.001>.

EINZIGER, G.; FRIEDMAN, R. TinyLFU : A Highly Efficient Cache Admission Policy. In: **Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. Torino: IEEE, 2014. p. 146–153. Available from Internet: <http://dx.doi.org/10.1109/PDP.2014.34>.

GAO, L. et al. Improving availability and performance with application-specific data replication. **IEEE Transactions on Knowledge and Data Engineering**, v. 17, n. 1, p. 106–120, 2005.

GAWADE, S.; GUPTA, H. Review of Algorithms for Web Pre-fetching and Caching. **International Journal of Advanced Research in Computer and Communication Engineering**, v. 1, n. 2, p. 62–65, 2012. Available from Internet: <www.ijarcce.com>.

GHANDEHARIZADEH, S.; YAP, J.; BARAHMAND, S. **COSAR-CQN : An Application Transparent Approach to Cache Consistency**. 2012. Available from Internet: <http://dblab.usc.edu/users/papers/cosarcqntr.pdf>.

GLASER, B. **Basics of grounded theory analysis.** [S.l.: s.n.], 1992. 128 p.

GLASER, B. G.; STRAUSS, A. L. The discovery of grounded theory. **International Journal of Qualitative Methods**, v. 5, p. 1–10, 1967. Available from Internet: <http://dx.doi.org/10.2307/588533>.

GUERRERO, C.; JUIZ, C.; PUIGJANER, R. Improving web cache performance via adaptive content fragmentation design. In: **Proceedings of the IEEE International Symposium on Network Computing and Applications**. IEEE, 2011. p. 310–313. Available from Internet: <http://dx.doi.org/10.1109/NCA.2011.55>.

GUO, P. J.; ENGLER, D. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In: **Proceedings of the 2011 International Symposium on Software Testing and Analysis**. New York, New York, USA: ACM Press, 2011. p. 287–297. Available from Internet: <http://doi.acm.org/10.1145/2001420.2001455>.

GUPTA, P.; ZELDOVICH, N.; MADDEN, S. A trigger-based middleware cache for ORMs. In: **Proceedings of the 12th ACM/IFIP/USENIX International Middleware Conference**. Lisbon, Portugal: Springer Berlin Heidelberg, 2011. v. 7049 LNCS, p. 329–349. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-25821-3_17>.

HUANG, J. et al. A browser-based framework for data cache in web-delivered service composition. In: **Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2010**. IEEE, 2010. p. 1–8. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5707138>.

HUEBSCHER, M. C.; MCCANN, J. a. A survey of autonomic computing—degrees, models, and applications. **ACM Computing Surveys**, ACM, v. 40, n. 3, p. 1–28, aug 2008. Available from Internet: <http://dl.acm.org/citation.cfm?id=1380584.1380585>.

INFANTE, A. Identifying caching opportunities, effortlessly. In: **Companion Proceedings of the 36th International Conference on Software Engineering**. New York, New York, USA: ACM Press, 2014. p. 730–732. Available from Internet: <http://dx.doi.org/10.1145/2591062.2591198>.

JORGENSEN, M. Evidence-based guidelines for assessment of software development cost uncertainty. **IEEE Transactions on Software Engineering**, IEEE, v. 31, n. 11, p. 942–954, nov 2005. Available from Internet: <http://dx.doi.org/10.1109/TSE.2005.128>.

JURISTO, N.; MORENO, A. M.; SANCHEZ-SEGURA, M. I. Guidelines for eliciting usability functionalities. **IEEE Transactions on Software Engineering**, IEEE, v. 33, n. 11, p. 744–758, nov 2007. Available from Internet: <http://dx.doi.org/10.1109/TSE.2007.70741>.

KICZALES, G. et al. Aspect-oriented programming. **European conference on object-oriented programming**, v. 1241/1997, n. June, p. 220–242, 1997. Available from Internet: <http://www.springerlink.com/index/X535M642082K783R.pdfhttp://www.ncbi.nlm.nih.gov/pubmed/22096456%5Cnhttp://www.ncbi.nlm.nih.gov/pubmed/22089444%5Cnhttp://link.springer.com/10.1007/BFb0053381>.

LABRINIDIS, A. Caching and Materialization for Web Databases. **Foundations and Trends® in Databases**, Now Publishers Inc., v. 2, n. 3, p. 169–266, mar 2009. Available from Internet: <http://dl.acm.org/citation.cfm?id=1754452.1754453http://dl.acm.org/citation.cfm?id=1754453http://www.nowpublishers.com/article/Details/DBS-005>.

LALANDA, P.; MCCANN, J. a.; DIACONESCU, A. **Autonomic Computing. Principles, Design and Implementation**. 1. ed. Springer-Verlag London, 2013. XV, 288 p. Available from Internet: <http://dx.doi.org/10.1007/978-1-4471-5007-7>.

LARSON, P.; GOLDSTEIN, J.; ZHOU, J. MTCache: Transparent mid-tier database caching in SQL server. **Proceedings of the International Conference on Data Engineering**, IEEE Computer Society, v. 20, p. 177–188, mar 2004. Available from Internet: <http://dx.doi.org/10.1109/ICDE.2004.1319994>.

LI, L. et al. An adaptive caching mechanism for Web services. In: **Proceedings of the International Conference on Quality Software**. IEEE, 2006. p. 303–310. Available from Internet: <http://dx.doi.org/10.1109/QSIC.2006.9>.

MA, H. et al. Paap. In: **Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval**. New York, New York, USA: ACM Press, 2014. p. 983–986. Available from Internet: <http://dx.doi.org/10.1145/2600428.2609490>.

MAPLESDEN, D. et al. Subsuming Methods. In: **Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering - ICPE '15**. New York, New York, USA: ACM Press, 2015. p. 175–186. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2668930.2688040>.

MERTZ, J.; NUNES, I. A Qualitative Study of Application-level Caching. **IEEE Transactions on Software Engineering**, v. 43, p. 20, 2016. Available from Internet: <https://doi.org/10.1109/TSE.2016.2633992>.

MERTZ, J.; NUNES, I. Seamless and Adaptive Application-level Caching. In: **VI Workshop de Teses e Dissertações do CBSoft (WTDSoft 2016)**. Maringá, PR: [s.n.], 2016. p. 7. Available from Internet: <http://cbsoft.org/articles/0000/1243/CBSoft2016-WTDSoft.pdf#page=77>.

MERTZ, J.; NUNES, I. A Survey of Application-level Caching in Web Applications. **ACM Computing Surveys (CSUR)**, 2017. Submitted.

NADI, S. et al. Where do configuration constraints stem from? An extraction approach and an empirical study. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 8, p. 820–841, aug 2015. Available from Internet: <http://dx.doi.org/10.1109/TSE.2015.2415793>.

NAMOUN, A. et al. Exploring Mobile End User Development: Existing Use and Design Factors. **IEEE Transactions on Software Engineering**, IEEE, p. 1–1, 2016. Available from Internet: <http://dx.doi.org/10.1109/TSE.2016.2532873>.

NEGRÃO, A. P. et al. An adaptive semantics-aware replacement algorithm for web caching. **Journal of Internet Services and Applications**, v. 6, n. 1, p. 4, feb 2015. Available from Internet: <http://dx.doi.org/10.1186/s13174-015-0018-4>.

NERELLA, V. K. S. et al. Exploring optimization and caching for efficient collection operations. **International Journal on Automated Software Engineering**, Kluwer Academic Publishers, v. 21, n. 1, p. 1–38, nov 2013. Available from Internet: <http://dx.doi.org/10.1007/s10515-013-0119-x>.

NGUYEN, K.; XU, G. Cachetor: detecting cacheable data to remove bloat. In: **Proceedings of the 9th Joint Meeting on Foundations of Software Engineering**. New York, New York, USA: ACM Press, 2013. p. 268. Available from Internet: <http://dx.doi.org/10.1145/2491411.2491416>.

PAL, M. B.; JAIN, D. C. Web service enhancement using web pre-fetching by applying markov model. In: **Proceedings of the 4th International Conference on Communication Systems and Network Technologies**. IEEE, 2014. p. 393–397. Available from Internet: <http://dx.doi.org/10.1109/CSNT.2014.84>.

PODLIPNIG, S.; BÖSZÖRMENYI, L. A Survey of Web Cache Replacement Strategies. **ACM Computing Surveys**, ACM, v. 35, n. 4, p. 374–398, dec 2003. Available from Internet: <http://dx.doi.org/10.1145/954339.954341>.

PORTS, D. R. K. et al. Transactional Consistency and Automatic Management in an Application Data Cache. In: **Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation**. CA, USA: USENIX Association, 2010. p. 279–292.

RADHAKRISHNAN, G. Adaptive application caching. **Bell Labs Technical Journal**, v. 9, n. 1, p. 165–175, may 2004. Available from Internet: <http://dx.doi.org/10.1002/bltj.20011>.

RAMASWAMY, L. et al. Automatic fragment detection in dynamic web pages and its impact on caching. **IEEE Transactions on Knowledge and Data Engineering**, v. 17, n. 6, p. 859–874, jun 2005. Available from Internet: <http://dx.doi.org/10.1109/TKDE.2005.89>.

RAVI, J.; YU, Z.; SHI, W. A survey on dynamic Web content generation and delivery techniques. **Journal of Network and Computer Applications**, v. 32, n. 5, p. 943–960, sep 2009. Available from Internet: <http://dx.doi.org/10.1016/j.jnca.2009.03.005>.

ROBILLARD, M. P.; COELHO, W.; MURPHY, G. C. How effective developers investigate source code: An exploratory study. **IEEE Transactions on Software Engineering**, IEEE, v. 30, n. 12, p. 889–903, 2004. Available from Internet: <http://dx.doi.org/10.1109/TSE.2016.2532873>.

SAJEEV, G. P.; SEBASTIAN, M. P. Building a semi intelligent web cache with light weight machine learning. In: **2010 IEEE International Conference on Intelligent Systems, IS 2010 - Proceedings**. IEEE, 2010. p. 420–425. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5548373>.

SILLITO, J.; MURPHY, G. C.; De Volder, K. Asking and answering questions during a programming change task. In: **IEEE Transactions on Software Engineering**. IEEE, 2008. v. 34, n. 4, p. 434–451. Available from Internet: <http://dx.doi.org/10.1109/TSE.2008.26>.

SIVASUBRAMANIAN, S. et al. Analysis of Caching and Replication Strategies for Web Applications. **IEEE Internet Computing**, v. 11, n. 1, p. 60–66, 2007. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4061123>.

SOUNDARARAJAN, G.; AMZA, C. Using semantic information to improve transparent query caching for dynamic content web sites. In: **Proceedings of the International Workshop on Data Engineering Issues in E-Commerce**. IEEE, 2005. v. 2005, p. 132–138. Available from Internet: <http://dx.doi.org/10.1109/DEEC.2005.25>.

STULOVA, N.; MORALES, J. F.; HERMENEGILDO, M. V. Practical run-time checking via unobtrusive property caching. **Theory and Practice of Logic Programming**, Cambridge University Press, v. 15, n. 4-5, p. 726–741, sep 2015. Available from Internet: <http://www.journals.cambridge.org/abstract_S1471068415000344>.

SULAIMAN, S.; SHAMSUDDIN, S. M.; ABRAHAM, A. **A Survey of Web Caching Architectures or Deployment Schemes**. 2013. Available from Internet: <http://se.fc.utm.my/ijic/index.php/ijic/article/view/33>.

SULAIMAN, S. et al. Autonomous SPY: Intelligent web proxy caching detection using neurocomputing and particle swarm optimization. In: **Proceedings of the 6th International Symposium on Mechatronics and its Applications**. Sharjah: IEEE, 2009. p. 1–6. Available from Internet: <http://dx.doi.org/10.1109/ISMA.2009.5164846>.

Veena Singh Bhadauriya, Bhupesh Gour, A. U. K. Improved Server Response using Web Pre-fetching: A review. **International Journal of Advances in Engineering & Technology**, v. 6, n. 6, p. 2508–2513, 2013.

WANG, J. A survey of web caching schemes for the Internet. **ACM SIGCOMM Computer Communication Review**, ACM, v. 29, n. 5, p. 36, oct 1999. Available from Internet: <http://dl.acm.org/citation.cfm?id=505696.505701>.

WANG, W. et al. EasyCache: a transparent in-memory data caching approach for internetware. In: **Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware**. New York, New York, USA: ACM Press, 2014. p. 35–44. Available from Internet: <http://dx.doi.org/10.1145/2677832.2677837>.

WONG, K.-Y. Web cache replacement policies: a pragmatic approach. **Network, IEEE**, v. 20, n. 1, p. 28–34, jan 2006. Available from Internet: <http://dx.doi.org/10.1109/MNET.2006.1580916>.

XU, G. Finding reusable data structures. **Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12**, ACM, v. 47, n. 10, p. 1017, nov 2012. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2384616.2384690>.

ZHANG, M.; LUO, H.; ZHANG, H. A Survey of Caching Mechanisms in Information-Centric Networking. **IEEE Communications Surveys & Tutorials**, PP, n. 99, p. 1–1, 2015. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7080842>.

APPENDIX A — APPLICATION-LEVEL CACHING PATTERNS CATALOG

Our qualitative study allowed us to understand how developers deal with application-level caching in their applications, by explaining design, implementation and maintenance choices. Our findings and observations were used as foundation to the provision of practical guidance for developers with respect to caching. Based on our study, we derived caching patterns, which can be used by developers to help them design, implement and manage cache. Caching patterns are classified into categories, explained below. The proposed patterns are summarized in Table A.1.

Design. Support to design decisions associated with application-level caching.

Implementation. Support to implementation issues of application-level caching, by providing solutions and guidance at the code level.

Maintenance. Support to performance analysis and improvement of application-level caching.

Table A.1: Caching Pattern Classification.

Pattern	**Classification**	**Intent**
Asynchronous Loading	Implementation	Design a mediator to asynchronously deal with caching.
Cacheability	Design	Provide a reasoning process to decide whether to cache or not particular data.
Data Expiration	Design and Maintenance	Given cacheable content, provide a reasoning process to choose a consistency management approach based on data specificities.
Name Assignment	Implementation	Ensure a unique key and keep track of the content cached.

Next, we present in detail our patterns and their components, which comprise a template for a caching pattern catalog. These components are (i) a *classification*, (ii) the pattern *intent*, (iii) the *problem* involved, (iv) the *solution* proposed, and (v) an *example*.

A.1 Asynchronous Loading

Classification: Implementation

Intent

Design a mediator to asynchronously deal with caching.

Synchronous caching operations can affect client-response time and blocks the request processing thread.

Problem Under certain circumstances the cost of populating the cache is very expensive (data provided by third-party systems via web services or the result of a heavy calculation process, and others). In such a scenario whenever the cache is invalidated (by automatic expiration or invalidation) the request processing is blocked while getting fresh data, which affects client-response time.

Solution Load the cache asynchronously with a separate thread or by using a batch process.

Rules of thumb

- At initial population of the cache, mainly for large caches, load the cache asynchronously with a separate thread or by using a batch process.
- At runtime, when the cache is invalidated, repopulate it in a background thread and then hide the cost of data retrieval to the end-users.
- It is strongly recommended the use of a third-party library or frameworks which already provides cache basic operations in an async way. There are options for the most of programming languages and cache providers.

Example The method in the following code example shows an implementation of the Cache-aside pattern based on asynchronous processing. An object is identified by using an ID as the key. The `asyncGet` method uses this key and attempts to retrieve an item from the cache. If a matching item is found, it is returned. If there is no match in the cache, it should retrieve the object from a data store, adds it to the cache, and then returns it (the code that actually retrieves the data from the data store has been omitted because it is data store dependent). Note that the load from cache has a timeout, in order to ensure that the cache interaction do not block the processing.

```
// Get a cache client
// it can be a trird-party library or an implemented module
// this facade should provide at least get, set and remove methods
Cache cache = Cache.getInstance();

public List<Product> getProducts() {

    List<Product> products = null;
    Future<Object> f = (List<Product>) cache.asyncGet("products");
    try {
      // Try to get a value, for up to 5 seconds, and cancel if it
      // doesn't return
      products = f.get(5, TimeUnit.SECONDS);

    // throws expecting InterruptedException, ExecutionException
    // or TimeoutException
    } catch (Exception e) {
        // Since we don't need this, go ahead and cancel the operation.
        // This is not strictly necessary, but it'll save some work on
        // the server.  It is okay to cancel it if running.
        f.cancel(true);
        // Do other timeout related stuff
    }

    if (products == null) {
        products = getProductsFromDB();

        // updates into cache should not block the request
        // return the user request as soon as possible
        cache.asyncSet("products", products);
    }

    return products;
}

public Product getProduct(String id) {

        Product product = null;
        Future<Object> f = (Product) cache.asyncGet("product" + id);
        try {
          product = f.get(5, TimeUnit.SECONDS);
        } catch (Exception e) {
            f.cancel(true);
        }

        if (products == null) {
            product = getProductFromDB(id);

            // updates into cache should not block the request
            // return the user request as soon as possible
            cache.asyncSet("product" + id, products);
        }

        return product;
}
```

The code below demonstrates how to invalidate an object in the cache when the value is changed by the application. The code updates the original data store and then removes the cached item from the cache by calling the `asyncDelete` method, specifying the key.

The order of the steps in this sequence is important. If the item is removed before the cache is updated, there is a small window of opportunity for a client application to fetch the data (because it is not found in the cache) before the item in the data store has been changed, resulting in the cache containing stale data.

```
public void updateProduct(Product product) {
  updateProductIntoDB(product);
  cache.asyncDelete("products");

  //optionally, it is possible to update the data into cache
  cache.asyncSet("product" + id, product);
}

public void deleteProduct(String id) {
  deleteProductFromDB(id);
  cache.asyncDelete("products");
  cache.asyncDelete("product" + id);
}
```

A.2 Cacheability

Classification: Design

Intent Provide a reasoning process to decide whether to cache or not particular data.

Problem Cache has limited size, so it is important to use the available space to cache data that maximizes the benefits provided to the application. Otherwise, it can end up reducing application performance instead of improving it, consuming more cache memory and at the same time suffering from cache misses, where the data is not getting served from cache but is fetched from the source.

Solution Even though there are many criteria that contribute for identifying the level of data cacheability, there is a subset that would confirm this decision regardless of the values of the other criteria. Changeability is the first criterion that should be analyzed while selecting cacheable data, then usage frequency, shareability, computation complexity, and cache properties should be considered.

Figure A.1 expresses a flowchart of the reasoning process to decide whether to cache data, based on the observation of data and cache properties. All criteria are tightly related to the application specificities and should be specified by the developer.

Rules of thumb

(a) Despite being frequently used, user-specific data are not shareable and may not bring the benefit of caching, being usually avoided by developers. In this case, a specific session component is used to keep and retrieve user sessions.

(b) If the data changes frequently, it should not be immediately discarded from cache. An evaluation of the performance benefits of caching against the cost of building the cache should be done. Caching frequently changing data can provide benefits if slightly stale data is allowed.

(c) Expensive spots (when much processing is required to retrieve or create data) are bottlenecks that directly affect application performance and should be cached, even though it can increase complexity and responsibilities to deal with. Methods with high latency or that consists of a large call stack are some examples of this situation and opportunities for caching.

In addition, we list content properties that should be avoided, which do not convey the influence factors in a good way and lead to problems such as cache trashing.

(a) User-specific data. Avoid caching content that varies depending on the particularities of the request, unless weak consistency is acceptable. Otherwise, the cache can end up being fulfilled with small and less beneficial objects. As result, the caching component achieves its maximum capacity earlier and is flushed or replaced many times in a brief period, which is cache thrashing.

(b) Highly time-sensitive data. Content that changes more than is used should not be cached given that it will not take advantage from caching. The cost of implementing and designing an efficient consistency policy may not be compensate.

(c) Large-sized objects. Unless the size of the cache is large enough, do not cache large objects, it will probably result in a cache trashing problem, where the caching component is flushed or replaced many times in a short period.

Example We list some typical scenarios where data should be cached and also give explanations based on the criteria presented.

(a) Headlines. In most cases, headlines are shared by multiple users and updated infrequently.

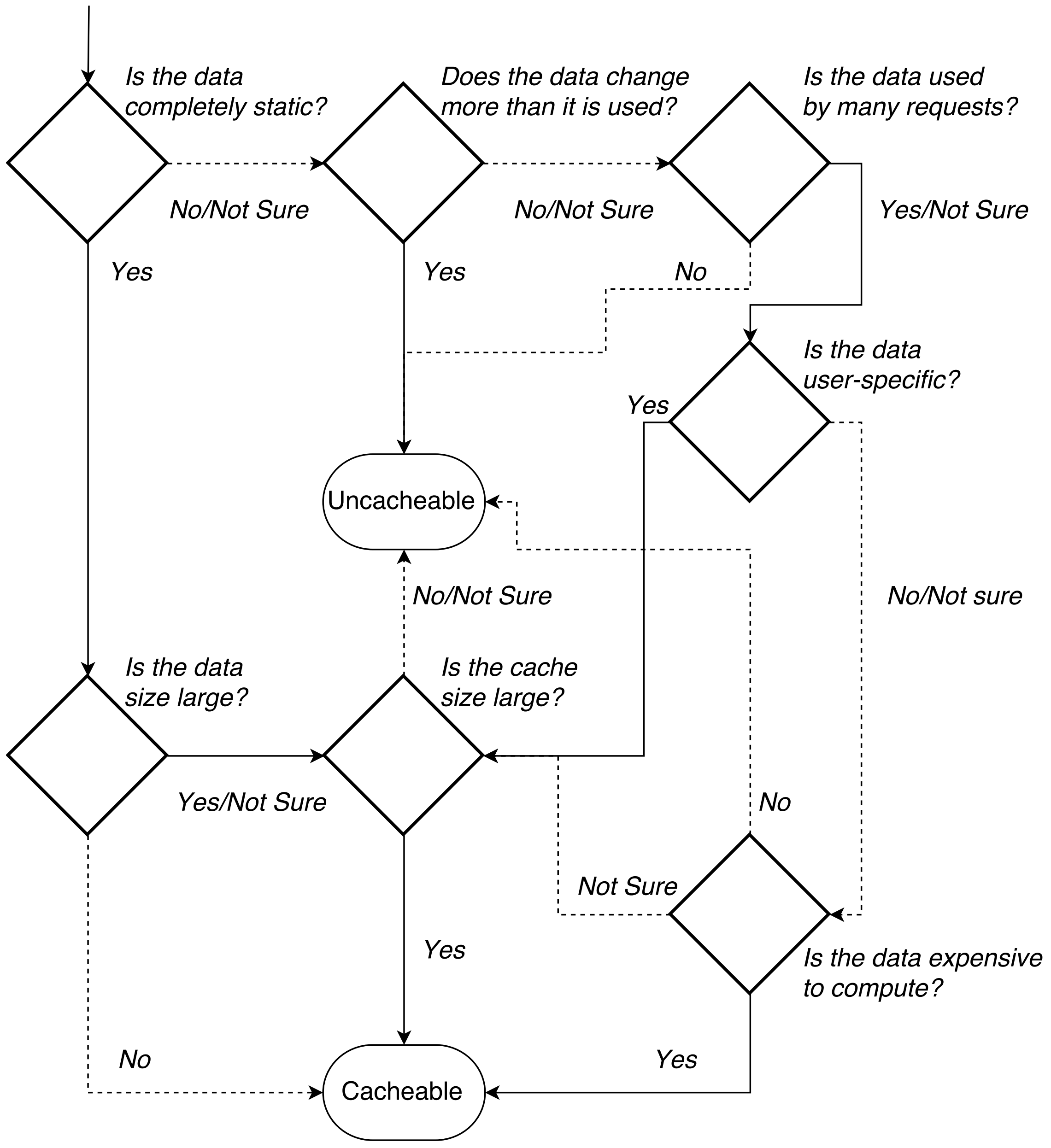
(b) Dashboards. Usually, much data need to be gathered across several application modules and manipulated to build a summarized information about the application.

(c) Catalogs. Catalogs need to be updated at specific intervals, are shared across the application, and manipulated before sending the content to the client.

(d) Metadata/configuration. Settings that do not frequently change, such as country/state lists, external resource addresses, logic/branching settings and tax definitions.

(e) Historical datasets for reports. Costly to retrieve or create and does not need to change frequently.

Figure A.1: Cacheability Flowchart.


Is the data completely static?
Does the data change more than it is used?
Is the data used by many requests?
No/Not Sure
No/Not Sure
Yes/Not Sure
Yes
Yes
No
Is the data user-specific?
Yes
Uncacheable
No/Not Sure
No/Not sure
Is the data size large?
Is the cache size large?
Yes/Not Sure
No
Not Sure
Is the data expensive to compute?
Yes
No
Yes
Cacheable

A.3 Data Expiration

Classification: Design and Maintenance

Intent Given the cacheable content, provide a reasoning process to decide a consistency management approach based on data specificities.

Problem It is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale. An inappropriate expiration policy may result in frequent invalidation of the cached data, which negates the benefits of caching.

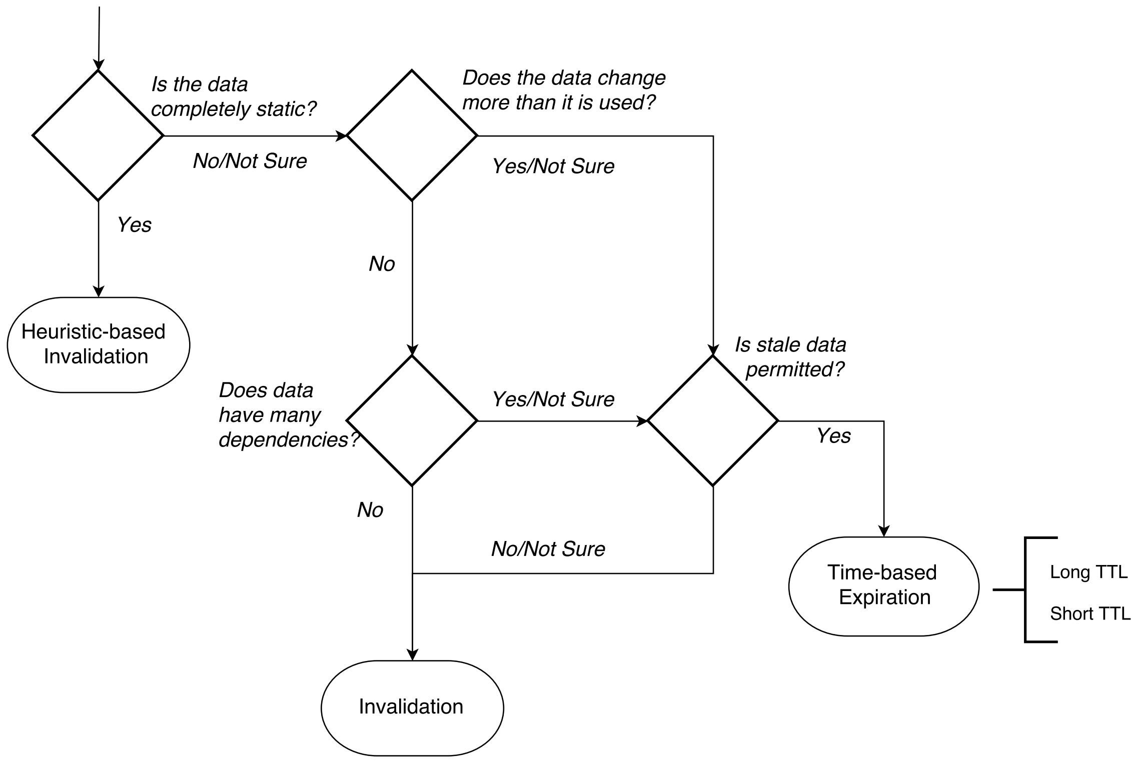
Solution Every piece of cached data is already potentially stale, and a good trade-off between performance benefits and cost of invalidation approaches should be achieved. Its necessary to determine the appropriate time interval to refresh data, and design a notification process to indicate that the cache needs refreshing. If the data is held too long, it runs the risk of using stale data, and if it was expired too frequently, it could affect performance.

Deciding on the expiration algorithm that is right for the scenario includes the following possibilities:

- Heuristic-based. Traditional algorithms such as least recently used (LRU) and least frequently used (LFU) can be used.
- Absolute expiration after a fixed interval. Expiration based on a pre-defined time-to-live (TTL), applied to every content.
- Invalidation. Caching expiration based on a change in an external dependency, such as modifications in the data by users actions.
- Flushing. Cleaning up the cache if a resource threshold (such as a memory limit) is reached.

Figure A.2 expresses a flowchart with the reasoning process to decide the appropriate consistency approach, based on observation of data properties. Changeability is the first property that should be analyzed while deciding, then staleness level and the amount of operations and dependencies related to the data should be considered. All properties are tightly related to the application specificities and should be defined by developer.

Figure A.2: Data Expiration Flowchart.


Is the data completely static?
No/Not Sure
Yes
Heuristic-based Invalidation
Does the data change more than it is used?
Yes/Not Sure
No
Is stale data permitted?
Does data have many dependencies?
Yes/Not Sure
Yes
No
No/Not Sure
Time-based Expiration
Long TTL
Short TTL
Invalidation

Rules of thumb

- While deciding the best consistency approach, it is important to measure the staleness degree and the lifetime of cached data.
- Frequently changed data is easily managed when associated with a TTL.
- Infrequently changed data provide more benefits when cached for long periods, thus manual invalidations or replacement are recommended.
- Determining how often is the cached information allowed to be wrong and work with weak consistency can be easier than defining a hard-to-maintain invalidation process. However, if the expiration period is too short, objects will expire too quickly, and it will reduce the benefits of using the cache. On the other hand, if the expiration period is too long, it risks the data becoming stale.
- Do not give all your keys the same TTLs, so they do not all expire at the same time. Doing this ensures that you do not get spikes of requests trying to make requests to your database because the cache keys have expired simultaneously.

- If the lifetime is dependent on how frequently the data is used, traditional heuristic-based eviction policies are right choices.
- If you frequently expire the cache to keep in synchronization with the rapidly changing data, you might end up using more system resources such as CPU, memory, and network.
- If the data does change frequently, you should evaluate the acceptable time limit during which stale data can be served to the user.
- Even if the data is quite volatile and changes, for example, every two minutes, the application can still take advantage from caching. For instance, if 20 clients are requesting the same data in a 2-minute interval, it is saving at least 20 round trips to the server by caching the data.
- Do not make the expiration period too short because this can cause applications to continually retrieve data from the data store and add it to the cache.
- Similarly, do not make the expiration period so long that the cached data is likely to become stale.
- Most caches adopt a LRU policy for selecting items to evict, but this may be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to help ensure that the cache is cost effective. It may not always be appropriate to apply a global eviction policy to every item in the cache. For example, if a cached item is very expensive to retrieve from the data store, it may be beneficial to retain this item in cache at the expense of more frequently accessed but less costly items.
- Cache services typically evict data on a LRU basis, but you can usually override this policy and prevent items from being evicted. However, if you adopt this approach, you risk your cache exceeding the memory that it has available, and an application that attempts to add an item to the cache will fail with an exception.

Example Consider a stock ticker, which shows the stock quotes. Although the stock rates are continuously updated, the stock ticker can safely be removed or even updated after a fixed time interval of some minutes.

A.4 Name Assignment

Classification: Implementation

Intent

- Ensure a unique key.
- Keep track of the content cached.

Problem Keys are important to keep track of the content cached while debugging or when it is necessary to invalidate and delete stale data from cache, in the case of changes in the source of information.

Solution When choosing a cache key, you should ensure that it is unique to the object being cached, and that it appropriately varies by any contextual values.

Rules of thumb:

- A unique key can be simple strings or more complex types like hashes, lists, or sets. The content can be identified by method signature and individual identification. Moreover, a tag-based identification should be used to group related content.
- If the object being cached relies on the current user (perhaps via HttpContext.Current.User), then the cache key may include a variable that uniquely identifies that user.

Example The key can be scoped to its particular function area, and be formatted with varying parameters.

```
var key = string.Format("MyClass.MyMethod:{0}:{1}", myParam1, myParam2);
```

```
class ShopCore extends ObjectModel
{
    public static function getCompleteListOfShopsID()
    {
        $cache_id = 'Shop::getCompleteListOfShopsID';

        if (!Cache::isStored($cache_id)) {
            $list = array();

            //database load
            Cache::store($cache_id, $list);
            return $list;
        }
    }
}
```

APPENDIX B — RESUMO ESTENDIDO

No contexto de aplicações web, requisitos de escalabilidade e performance estão sempre presentes. Cache é uma das soluções que podem suprir tais requisitos. Elas podem ser inseridas ao longo de toda a infraestrutura web, desde a fonte de informação (ex: base de dados) até o usuário final, no próprio navegador.

Uma das técnicas de cache em aplicações web é a cache em nível de aplicação, onde o desenvolvedor identifica operações ou computações com características cacheáveis como computações frequentes ou custosas e, ao invés de recomputar esse conteúdo a cada nova requisição, o resultado da operação é armazenado e reutilizado quando necessário.

No entanto, esse processo é essencialmente manual, uma vez que essa tarefa de identificação de oportunidades de cache é uma tarefa que depende do conhecimento de detalhes da aplicação, como o domínio e regras de negócio. Tal dependência também torna mais complexo a proposta de soluções para automatização ou abstração de tarefas de cache.

Diante disso, cache a nível de aplicação é uma tarefa desafiadora e que pode afetar várias etapas do desenvolvimento do software, tais como projeto, implementação e manutenção. Do ponto de vista de projeto, esse tipo de cache é essencialmente um esforço empírico dado que o desenvolvedor identifica oportunidades baseado em seu conhecimento dos detalhes da aplicação, implementa a lógica de caching necessária e por fim avalia se houveram ganhos com testes de performance. A implementação dessa lógica de caching por si só consiste em uma adição de código extra e que possivelmente estará entrelaçado com o código da aplicação (ex: regras de negócio), comprometendo a separação de responsabilidades. Por fim, diante da evolução da aplicação ou do perfil de acesso dos usuários, essa lógica de caching pode eventualmente se tornar obsoleta, não refletindo em benefícios na performance. Dessa forma, tal lógica também deve ser evoluída e revista constantemente, o que implica em um tempo adicional de manutenção. Nesse contexto, cache a nível de aplicação é uma tarefa que demanda tempo e esforço dos desenvolvedores além de ser extremamente suscetível a erros.

Diante de todos esses problemas, alguns frameworks e bibliotecas existentes são capazes de auxiliar em tal tarefa de caching, no entanto apenas questões de implementação são endereçadas por tais ferramentas, deixando todo o raciocínio de caching para o desenvolvedor. De fato, não existem padrões ou diretrizes para o desenvolvimento de uma cache eficiente. Em geral, desenvolvedores procuram melhores práticas ou tutoriais em sites especializados, no entanto tais práticas apresentam apenas experiências e não tem embasamento científico que suporte tais práticas. Do ponto de vista de automatização, em sua maioria, técnicas de caching existentes não são capazes de consideram detalhes da aplicação para guiar as decisões de cache. Ainda, tais técnicas em geral não exploram adaptação das decisões de cache diante de mudanças na aplicação, o que exige que o desenvolvedor continue por revisar as decisões de cache manualmente.

Diante disso, esse trabalho tem como questão de pesquisa: *Como auxiliar desenvolvedores durante o projeto, implementação e manutenção de cache a nível de aplicação em aplicações web de forma a reduzir o esforço demandado por tais atividades?*

Para isso, dentre os objetivos deste trabalho estão a proposta de diretrizes e padrões de projeto, implementação e manutenção de cache a nível de aplicação, de forma que desenvolvedores possam atingir uma implementação de cache eficiente a um menor custo de esforço e tempo. Além disso, este trabalho tem como objetivo a automatização de uma importante tarefa de caching, a identificação de oportunidades de caching.

Para atingir tais objetivos, primeiramente foi conduzido uma pesquisa qualitativa de modo que se pudesse entender como desenvolvedores atuam em relação a cache em suas aplicação atualmente. Dessa forma, é possível entender o conhecimento implícito e espalhado por essas aplicações, para que então esse conhecimento possa ser extraído, estruturado e documentado como soluções reutilizáveis. Para a execução dessa pesquisa, foram selecionadas 10 aplicações web de código aberto e comerciais com diferentes características de domínio, tamanho e linguagens de programação. Dessas 10 aplicações foram obtidos o código fonte, issues, documentação e, para a aplicações comercial, foi possível interagir com o desenvolvedor.

A partir de obtenção dessas fontes de informação, primeiramente foi realizada uma análise objetiva, com o intuito de verificar a contribuição das aplicações para o estudo e também para analisar o impacto da cache a nível de aplicação no código base das aplicações. A se destacar dessa análise, para a aplicação *open edX* 10.76% dos arquivos da aplicação continham lógica de cache. Para a aplicação *shopizer* 3.02% de todo o código fonte é especificamente para caching e, por fim, a partir da análise das issues foi constatado que 4.99% de todas as issues da aplicação *pencilblue* estão relacionadas a caching. Tais informações refletem o impacto da cache na aplicação.

Após a analise objetiva, foi então conduzido um estudo qualitativo o qual consistiu na análise manual das informações sobre as aplicações e observação de problemas, soluções, decisões ou quaisquer outras informações relevantes sobre cache. Tais observações eram então rotuladas de acordo com seu significado. Após a rotulação das observações na etapa inicial, esses rótulos eram comparados uns aos outros com o objetivo de unificá-los e então emergir uma categoria mais abstrata e representativa sobre aquele conceito. Com base na posterior análise de todas essas categorias derivadas do estudo, foi possível então derivar 16 diretrizes e 4 padrões, relacionados a questões de projeto, implementação e manutenção de soluções de cache a nível de aplicação.

As diretrizes correspondem a regras gerais ou recomendações para que desenvolvedores possam alcançar uma cache eficiente diante de um menor esforço. Já os padrões refletem soluções reutilizáveis para problemas recorrentes de cache. Padrões possuem uma estrutura fixa que consiste em uma classificação, um problema, uma intenção e, por fim, uma solução.

Como exemplo de diretrizes derivadas está a convenção de nomes para chaves, uma vez que a cache em sua essência é uma estrutura de chave-valor e apenas a chave identifica tal conteúdo. Nesse sentido, a recomendação é o estabelecimento de uma convenção de nomes, como por exemplo relacionado ao contexto, a informação em si ou a localização de tal lógica de cache. Dessa forma é possível garantir que as chaves sejam únicas, além de facilitar o a localização da lógica de cache, em situações de debug, por exemplo. Uma diretriz de projeto é a avaliação de fronteiras da aplicação. Tais fronteiras em geral representam pontos do sistema onde são realizadas chamadas ou interações com componentes externos, tais como bases de dados, serviços web ou até outras aplicações. Essa comunicação com outros sistemas envolve geralmente um custo de rede, além de que o sistema que está sendo consultado muitas vezes não permite avaliar o que está sendo processado, o que leva a constantes gargalos na aplicação. Dessa forma, ao iniciar o raciocínio sobre o que deve ser cacheado, o desenvolvedor deve primeiramente avaliar essas comunicações. Por fim, uma diretriz de manutenção refere-se a documentação precisa de avaliações de desempenho das soluções de cache. Comumente, desenvolvedores implementam lógicas de cache e então avaliam se tal lógica reflete em benefícios para o desempenho da aplicação. Tais avaliações em geral não são bem documentadas, e dessa forma tornam-se irreprodutíveis e podem levar a conclusões precipitadas sobre a lógica de cache proposta. Assim, a recomendação é a documentação precisa da avaliação, desde características de hardware utilizadas, até informações manipuladas. Dessa forma é possível que outras pessoas possam reproduzir e avaliar com maior precisão o benefício de uma solução de cache.

Quanto a padrões, um dos 4 padrões derivados corresponde ao padrão de cacheabilidade. Este padrão é classificado como padrão de projeto e tem como objetivo auxiliar o desenvolvedor na decisão sobre qual conteúdo deve ser cacheado. O problema em questão se refere ao espaço limitado de cache, e a necessidade de manter apenas o conteúdo mais relevante nesse espaço. Por relevância, entende-se aquele conteúdo que mais beneficia o desempenho da aplicação, se cacheado. Para isso, eu proponho um processo de raciocínio que consiste na avaliação de uma sequencia de critérios para a decisão se o conteúdo deve ou não ir para a cache.

Esse padrão é apresentado em forma de um fluxograma onde cada decisão no fluxo refere-se a um questionamento que o desenvolvedor deve fazer sobre a informações sendo avaliada. Esse fluxograma apresenta um conjunto de critérios mais relevantes e também um ordem de precedência a qual deve ser seguida. Tais critérios consistem em: estaticidade, mudança, frequência, compartilhamento, custo e tamanhos, de cache e conteúdo. Este padrão possibilita uma decisão rápida e eficiente para o desenvolvedor.

Tais padrões e diretrizes por si só já oferecer suporte ao desenvolvedor nas atividades de desenvolvimento de uma solução de cache, levando a um componente eficiente com um menor tempo e esforço. Uma vez que diretrizes e padrões são observados, o próximo passo é então automatizar tais soluções, abstraindo ainda mais as responsabilidades de cache para o desenvolvedor.

Diante disse, eu proponho a automatização de uma importante tarefa de cache, a decisão sobre oportunidades de cache. Nesse contexto eu proponho uma abordagem transparente e automatizada capaz de monitorar a aplicação em tempo real, analisar e encontrar oportunidades de cache e cachear as melhores oportunidades. Todo esse processo considerando informações específicas da aplicação, que de fato é a informação utilizada pelos desenvolvedores durante o processo de projeto de um solução de cache. Além disso, a solução é totalmente baseada na observação da carga de trabalho, levando a uma solução específica e otimizada para a aplicação.

Tal abordagem consiste em 3 atividades, monitoração de traços de execução, análise de cacheabilidade e gerenciamento da aplicação e cache. Essas atividades são executadas diante de 2 ciclos, um proativo e um reativo, os quais são executados em background e em tempo real, respectivamente. A primeira atividade, monitoração de traços de execução, consiste em obter informações da execução da aplicação, para isso a abordagem monitora cada chamada de método, e para cada ocorrência, um traço dessa execução é armazenado. Para isso, programação orientada a aspectos é utilizada para interceptar execuções de métodos no inicio e fim, com o intuito de capturar a chamada, com seus devidos parâmetros, e o retorno da chamada. Além de informações da chamada de método, outras informações específicas da execução de tal método são armazenadas, tais como o tempo de execução e o usuário em sessão no sistema. Além de informações da execução da aplicação, o comportamento da cache em termos de estatísticas da cache são monitoradas. Essas estatísticas são obtidas a partir do próprio componente de cache e apresentam informações tais como a quantidade de espaço utilizado e livre, hits e misses. Toda essa informações é monitorada dentro de um intervalo de tempo, um snapshot do sistema. E então armazenadas em uma base de dados.

A segunda etapa da abordagem consiste na análise dessas informações anteriormente coletadas. Tal análise é realizada em background, não afetando a execução do sistema. Nessa etapa, é avaliada a cacheabilidade dos métodos monitorados. Tal avaliação é baseada no padrão de cacheabilidade apresentado anteriormente. Em uma situação manual, o desenvolvedor deveria instanciar e responder cada questionamento do padrão para atingir uma resposta, cacheável ou não cacheável. Na abordagem, cada decisão do padrão de cacheabilidade foi transformado em uma métrica, calculada a partir dos traços de execução monitorados da etapa anterior. Com base nessas métricas cada ponto da decisão cacheabilidade é respondido de acordo com o padrão, no entanto de forma automatizada e automática. Por exemplo, para a decisão de estaticidade, é verificado nos traços de execução quais foram os métodos onde dado uma sequencia de parâmetros específica, o método sempre retornou o mesmo valor, o que significa que o método é estático. Uma vez calculados todos as métricas para os métodos monitorados, é possível então identificar quais chamadas de métodos devem ou não serem cacheadas e um conjunto de chamadas cacheáveis é derivado.

Por fim, novamente em tempo de execução, a etapa de gerenciamento da cache e acionada com base na lista de chamadas cacheáveis identificada na etapa anterior. Uma vez que um método cacheável é invocado, a implementação baseada em aspectos realiza o cache de tal informações e reutiliza posteriormente.

Para avaliar a abordagem, eu proponho uma comparação com as decisões dos desenvolvedores, de modo a avaliar o comportamento de uma cache automatizada com uma cache desenvolvida manualmente, por desenvolvedores. Para isso, eu realizei uma implementação baseada na linguagem Java, e proponho um experimento com o objetivo de avaliar os ganhos de desempenho que a cache proporciona para aplicação e também uma investigação sobre quais as oportunidades encontradas pelos desenvolvedores e quais encontradas pela abordagem automatizada. Outras questões de cache como algoritmos de substituição, tamanho de cache e regras de consistência foram abstraídas e então solucionadas com as mesmas decisões dos desenvolvedores ou tomando valores de referencia e bem aceitos pela comunidade.

Para a realização desse experimento, eu selecionei 3 aplicações de diferentes tamanhos e domínios. Para cada aplicação, foram criados três versões: sem nenhuma lógica de cache, com a lógica manual e proposta pelos desenvolvedores, e por fim com a abordagem automatizada. Então, as três variações de cada aplicação foram submetidas a testes de estresse onde usuários simulados executas requisições frequentes para as aplicações. A execução e comportamento das aplicações foram avaliados diante de duas métricas, throughput e hit ratio, que referem-se a quantidade de requisições em que uma aplicação é capaz de responder em um intervalo de tempo e taxa de acessos a cache que resultaram em um hit, respectivamente.

Os resultados desse experimento demonstram que a abordagem automatizada sempre apresenta benefício de performance em relação a aplicação sem cache, e no mínimo, tal benefício é equivalente a cache manual. Além disso, para duas aplicações a cache automatizada é superior a cache manual em benefício de desempenho. No entanto, ao analisar o hit ratio, a cache automatizada apresentou um comportamento distinto. Para uma aplicação não houve mudança, no entanto, para as duas restantes, houve uma variação onde em um caso houve um aumento significativo e noutro caso uma redução significativa no hit ratio. Para entender esse comportamento foi realizado uma investigação manual e então identificado que para o caso de aumento do hit ratio, os desenvolvedores faziam cache de operações de busca, as quais se baseiam numa string. No entanto, todas as operações de busca eram cacheadas na cache manual, diferentemente da cache automatizada a qual é capaz de distinguir operações que de fato são relevantes e reutilizáveis de outras não relevantes. Para o caso onde houve uma redução do hit ratio, isso se deve a identificação de muitas oportunidades por parte da cache automatizada, e uma vez que o tamanho da cache foi escolhido como o mesmo utilizado pelo desenvolvedor, todo o conteúdo cacheável não era levado até a cache, o que resultou em uma alta taxa de misses, reduzindo o hit ratio. Por fim, é possível observar que conforme a execução da aplicação, a mesma tende a responder mais requisições em um menor período de tempo com a cache automatizada em relação aos outras configurações, manual e sem cache.

Além da análise de desempenho das aplicações, foi conduzido uma análise das oportunidades identificadas por cada uma das abordagens, manual e automatizada. Nessa análise foi possível identificar que para todos os métodos cacheados pelo desenvolvedor a abordagem automatizada foi capaz de encontrar oportunidades de cache. Isso se deve a cautela com que o desenvolvedor seleciona oportunidades de cache. Ao transformar a decisão em métricas objetivas, tais oportunidades se tornam ainda mais evidentes e sempre serão cacheadas. No entanto, diversas outras oportunidades deixadas pelos desenvolvedores foram encontradas pela abordagem automatizada.

Nesse contexto, a abordagem automatizada proposta foi capaz de identificar oportunidades de cache com uma melhora de desempenho em relação a solução de cache manual, além de ser totalmente baseada na carga de trabalho da aplicação e nos detalhes da mesma. No entanto, o overhead esperado do monitoramento de todas as chamadas de métodos possui um impacto na aplicação e pode levar uma redução nos ganhos de desempenho proporcionados pela cache automatizada. Além disso, uma limitação da abordagem é que uma vez que apenas as entradas e saídas de uma chamada de método são observados, todos os detalhes de execução do método são ignorados. Dessa forma, se métodos que realizam alterações no estado da aplicação, como por exemplo instanciam threads ou alterar variáveis estáticas, uma vez cacheados, não executarão tal operação esperada, possivelmente levando a aplicação a um estado inconsistente. No entanto, tais métodos são facilmente identificáveis, e a implementação da abordagem permite que desenvolvedores indiquem tais métodos por meio de anotações no código. Assim, métodos que não devem ser cacheados são desconsiderados pela abordagem.

De maneira geral os resultados desse trabalho apresentam um avanço na pesquisa de cache a nível de aplicação, provendo uma melhor experiência para os desenvolvedores e auxiliando na tarefa de desenvolvimento de uma solução de cache eficiente, com um menor esforço e tempo. Trabalhos futuros consistem na avaliação de soluções para as limitações da abordagem automatizada e também na proposta de automatização de outras tarefas e decisões de cache, baseando-se nas diretrizes e padrões derivados e que ainda não foram explorados.