Developer's Cognitive Effort Maintaining Monoliths vs. Microservices - An Eye-Tracking Study

1st Georg Simhandl, 3rd Uwe Zdun
University of Vienna, Faculty of Computer Science,
Research Group Software Architecture
Vienna, Austria
firstname.lastname@univie.ac.at

2nd Philipp Paulweber
fiskaly GmbH
Vienna, Austria
ppaulweber@fiskaly.com

***Abstract*—The microservices architectural style improves flexibility and performance but might come at the cost of complexity and cognitive effort. Understanding how architectural design decision affects cognitive effort can support software engineers in designing and building more maintainable systems. However, little research exists measuring the impact of architectural styles on developers' cognitive processes. To the best of our knowledge, no empirical study on navigation and comprehension strategies compares a microservice-based and monolithic variant of a trading system, which is widely used for empirical research on information system evolution. To assess the cognitive processes, we conducted an eye-tracking study (n=42) of typical maintenance tasks. We randomly assigned participants to two groups performing a feature location and a code change task using the trading system's monolithic or microservices-based implementation. Efficacy is significantly higher in the monolith variant. We investigate the reasons and propose a cognitively-grounded method to analyse comprehension strategies and estimate maintenance effort. We measure differences in comprehension strategies (top-down and bottom-up comprehension) and conclude that 1) there is a significant difference in visual effort and time taken to identify a specific feature, and 2) bottom-up comprehension is more frequently applied in microservices than in the monolith variant. Finally, we discuss our findings and their implications for practitioners and the research community.**

***Index Terms*—software architecture, microservices, monolith, eye-tracking**

I. Introduction

Organisations are increasingly adopting the microservices architectural style, which has the potential to overcome the deficiencies of earlier approaches and design, develop and deploy using modern paradigms, reflect the organisational structure and improve flexibility and performance [1, 2]. Adopting a new architectural style is multi-faceted and comes with risks and costs software engineers must consider. For most applications that do not face thousands of concurrent users, the foundational architectural decision is a trade-off between the costs and benefits of a monolith or (the migration to) microservices. In microservices, the modularity comes at the cost of performance due to remote calls or adds new complexities mitigating those performance losses, e. g., by using asynchrony or request bundling. Recent studies empirically evaluate the performance and scalability of monoliths compared to microservices, e. g., providing insights into trade-offs between monoliths or microservices [3]. Another source of information that companies can use to decide whether to migrate to microservices or not is provided by Auer et al. [4]. They discover that the most commonly mentioned motivation to migrate to microservices is to improve the maintainability of the system [4]. As software engineers spend most of their time reading and understanding source code during maintenance, a critical decision driver is *cognitive cost*[1]. There are many studies on cognitive load during program comprehension, and the effect of e. g., source code lexicon and readability on the cognitive load is well documented [5]. Yet, there is no empirical study on developers' comprehension of microservices in the context of software maintenance. Although there is vast research on program comprehension models, i. e., how developers form a mental model of a program, there is a gap in understanding modern architectural styles like microservices, calling for an update to a *contemporary model of program comprehension* [6]. Siegmund [6] in this regard mentions that developers perform program-comprehension tasks that go beyond the source-code level and include understanding *overall software architecture*, understanding *type structures and call hierarchies* as well as the *relationship between components*. This study aims to contribute to a contemporary program comprehension model focussing on two developers ' main tasks, i. e., the understanding of *call hierarchies and the relationship between components* and measuring the *cognitive effort involved in maintenance* of microservices in comparison to monolithic architectures.

Specifically, we focus on the following research questions:

- **RQ1**: How do developers comprehend call hierarchies in the context of feature location tasks in microservices and monolith architectural styles?
- **RQ2**: What is the cognitive effort to navigate and maintain microservices compared to monoliths?

Since program comprehension as a set of cognitive processes is performed at the speed of the eye [7], we build upon prior work in program comprehension research and apply eye-tracking. We report on a controlled experiment (Section IV) assessing developers' cognitive processes maintaining microservices (treatment group) and monoliths (control group) using extensions of a monolith and its migration to microservices of a trading system, an example system widely used in

[1] https://martinfowler.com/articles/microservice-trade-offs.html

empirical software engineering (Section IV-D). We describe the experimental design adhering to guidelines for controlled experiments [8] using eye-tracking [9, 10]. We measure task performance, i.e., response time and correctness, assess developers' comprehension strategies and comprehension models (Section IV-G), and measure developers' cognitive effort performing typical maintenance tasks (Section IV-E). We report the results in Section Section V, conclude with implications and discuss future directions to build a cognitive-grounded decision model for architectural decisions (Section VI).

II. Related work

The architectural style is a set of profound decisions on design elements and formal arrangements to increase comprehensibility [11, 12]. To understand the implications of microservices architectural style and provide guidance for practitioners in decision-making, studies compare monolith and microservices architectural styles. Most comparative studies focus on performance-related attributes [3, 13]. Recent empirical studies investigate architectural decisions, e.g., focusing on migration from monoliths to microservices [14]. In another Grounded Theory study Auer et al. [4] discovered that the most common motivation to migrate to microservices was to improve the system's maintainability and reduce overall system complexity, which is frequently associated with maintenance. However, only a few studies measure and compare the maintenance effort of microservices and monoliths [4].

Software maintenance tasks require developers to spend considerable time searching and navigating source code to understand the parts of the system relevant to the change task. Prior studies find that developers spend much time fixing bugs and making code more maintainable [15, 16]. In a recent field study on 78 professional developers Xia et al. [17] report that developers spend ∼58% of their time on program comprehension activities, followed by navigation (∼ 24%), others (13.40%), and only ∼5% editing.

Substantial previous research has investigated program comprehensibility surrounding mental representations of programs and related cognitive models [18]. During navigating and understanding, and then editing code, developers implicitly build code context models that consist of the relevant code elements and the relations between these elements. These models are characterised by a high lexical and structural cohesion [19]. Recent studies provide detailed views on developers' cognition using eye tracking, investigating drivers for readability and reading order [20–22] and cognitive models of a structural understanding of source code, e.g. by studying eye movement during summarisation tasks [23, 24]. These works tend to focus on code snippets or small programs. Another stream of eye-tracking in program comprehension research focuses on the understandability of software design aspects, e. g., design patterns and class relationships recovered from source code or inspecting UML diagrams Guéhéneuc [25], Jeanmart et al. [26], Sharif and Maletic [27]. Wang et al. [28] report a study on the feature location process. Their study suggests that the feature location process can be understood hierarchically on a set of actions a developer performs, e. g., reading and searching source code or exploring dependencies, the phase level, e. g., *seed search, extend and validate* phases, and on a pattern level. In the crucial *seed search phase*, Wang et al. [28] identified three search patterns, namely the information-retrieval-based, execution-based and exploration-based search patterns. Only a few works measure developers' cognitive strategies in larger systems [20, 29]. However, there is a lack of approaches providing details on the impact of architectural decisions on developers' cognitive effort and the related cognitive strategies in building a mental model of the program.

To the best of our knowledge, there is no study comparing monoliths and microservices from a cognitive perspective using eye-tracking, yet.

III. Cognition models

Cognition models in program comprehension describe the cognitive process and temporary information structures in the developer's head that form a mental model of a software system. The cognitive processes use existing knowledge to obtain new knowledge and build a mental model of the program under consideration [30, 31]. Von Mayrhauser and Vans [30] provide an integrated metamodel describing how developers create mental representations of programs. This metamodel includes four components, the *top-down comprehension*, used when the code or the programming language is familiar and the *program model*, which is invoked when the system or the code is unfamiliar. The *situational model* describes data-flow abstractions in the program. These three models rely on the fourth component, the *knowledge base* representing the developers' current knowledge and is used to store and create new knowledge [30].

Top-down comprehension refers to the process when a developer has a hypothesis of the program and subsequently locates source code elements to confirm this hypothesis. Developers read more closely and apply bottom-up comprehension to form hypotheses for the program by mentally grouping each set of statements into higher-level abstractions. These abstractions are aggregated further until a high-level understanding of the program is obtained [30, 31].

Recent studies on program comprehension evaluate specific aspects of cognition models using eye-tracking. Abid et al. [24] compare novices' and experts' cognition models, concluding both experts and novices read the methods more closely, using the bottom-up mental model rather than bouncing around using top-down comprehension strategies. Priorly, Rodeghero and McMillan [23] performed a study on code summarisation with professional developers, showing that developers read the source code differently than natural language text, skimming and jumping between source code elements, in contrast to more thorough techniques, such as close and sectional reading, and conclude that programmers not only follow specific patterns when they read and summarise source code, but they also all tend to use similar patterns compared to each other. The effect of reading order is also studied by Peitek et al. [22]. They

conclude that the source code's linearity strongly affects the reading order, while the comprehension strategy has only a minor effect [22]. In a recent study, Sharafi et al. [29] found that successful developers' navigation strategies are quantifiably different. The frequent switching of *Areas of Interests* (AOI), also called *thrashing* is associated with worse performance [29].

A. Comprehension process

To determine the type of mental model Abid et al. [24] proposed to divide each method into *chunks* and subsequently analyse the comprehension strategies such as bottom-up comprehension, which is characterised by *chunking* of these source code elements. We extend this approach and take all source code elements of a *Java* source code files, i. e., including method signature and annotation, class and field declarations, constructor declarations and import statements into account. As developers tend to use documentation to apply different comprehension strategies, we kept *JavaDoc* and *Readme* files in the code repository. Instead of manually selecting these sections, we apply an automated *extraction* approach. We parse the abstract syntax tree (AST) and automatically label logically connected source code elements. The extractor also partitions and labels method bodies into sections containing control or data flow, associated statements, and separate sections containing comments or blank lines. The rationale behind this extraction rule is that developers tend to place comments before the code. Blank lines separate are also a visual aid to separate different tasks [32].

To determine the comprehension strategies using scan path analysis, Abid et al. [24] define two detection rules: If a developer starts reading a section and the next line read belongs to the same section, they conclude that the developer is reading closely and performing *bottom-up comprehension*. They interpret switches to a different source code section as *top-down comprehension* [24]. We extend this approach beyond code snippets to larger multi-component systems based on prior empirical studies [23, 24, 29].

B. Regression

Regressions indicate that a participant had to revisit a previous part, possibly due to an insufficient understanding or following the execution flow [21, 22]. Backward movements in the text are called regressions. Few regressions and short fixations characterise good readers. Difficult texts usually induce longer fixations, short saccades and frequent regressions [21]. Busjahn et al. [21] define two gaze-based measures for source code reading, the *regression rate* (percentage of backward saccades of any length) and *line regression rate* (percentage of backward saccades within a line).

IV. Experimental Design

To investigate the cognitive effects of architectural styles, we designed a controlled experiment using the code base of a *trading system* implemented in a monolith (control group) and the migration of this system to microservices (treatment group) as the stimuli. Following the guidelines of Jedlitschka et al. [8] for controlled experiments in software engineering research adhering to Holmqvist et al. [10] for eye-tracking studies, we report on the participants involved, the eye-tracking equipment, instructions and tasks given to participants, quantitative attributes of recorded eye-tracker signals, fixation detection algorithms and AOI analysis. Finally, we report on exclusion criteria and the pre- and post-recording.

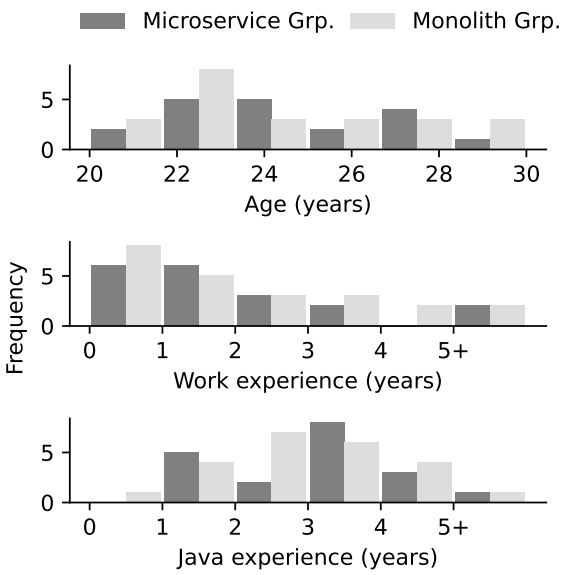

Figure 1. Participant's age, experience working in the software industry and Java programming experience (years) per treatment (microservice) and control (monolith) group.

A. Participants and Recruiting Process

We recruited 44 students enrolled in software engineering courses offered at the end of their B.Sc. and the beginning of their M.Sc. programs at the author's institution. As the study focuses on typical interviewing or onboarding tasks, when joining a new software team, students finishing or graduating computer science studies as participants are valid representatives for early-stage software professionals in empirical software engineering experiments. All participants gained experience designing and developing microservices, at least throughout the course. Besides a brief announcement and information about the eye-tracking study's goal and procedure, participants did not get further training or preparation. Participating students were rewarded with extra course credits as an incentive. Two participants didn't complete the experiment, and the incomplete data was removed from the analysis. Of the remaining 42 participants, ten were female, and 32 were male. The mean age of participants in the treatment group (microservices) is similar (24 years, ± 2.43) to the control group (monoliths), where the mean age is also 24 (± 2.81).On average, participants gained 1.8 years of work experience in the software industry and have intermediary knowledge in Java language (see Figure 1).

B. Setup and Equipment

The eye-tracking experiment took place in the laboratory rooms of the faculty, simulating an undistracted and realistic office environment and allowing stable conditions throughout the whole study. We used *Pupil Labs Core* head-mounted monocular eye-tracker to record eye gaze position, pupil dilation and blink rate and duration. The eye-tracker records gaze behaviour at 200Hz with a pupil camera. The frontal world camera with a resolution of 1280x720 pixels is directed towards a 27-inch monitor screen with a resolution of 2560 by 1600 pixels, representing the participant's view on the screen. We chose Visual Studio Code (VSCode) as the integrated development environment (IDE) for source code reviewing and editing tasks. VSCode supports syntax highlighting code completion, aids in navigating source code, and is extensible using various plugins and extensions. It is generally possible to implement custom extensions using the VSCode API. At the time of writing, the API didn't offer access to the viewport of the code editing windows, allowing to record the source code or text visible to the user. We implemented a simple extension tracking mouse position, scroll, and keyboard events to determine the visible text lines. We used a screen background with rectangular visual markers to enable gaze-to-surface mapping. The IDE was positioned at the centre of the screen, scaled to about 24 inches, and surrounded by eight visual markers.

C. Procedure

We used a randomised design to assign control and treatment groups participants. We conducted an ex-ante survey on demographics such as age, gender, education level, and software engineering experience in terms of years of development and modelling expertise. We asked for the motivation to develop software.

Following the experiment protocol, we calibrated the eye-tracker and started recording the screen and Microsoft Windows 10 system events, VSCode extension recording mouse and keyboard, and source code file opening and switching events. After the experiment, we conducted a post-hoc survey on the tasks' self-estimated difficulty. We finished the survey with an open question about how IDEs could be improved to support program comprehension.

The experimental setup was pilot tested with three software engineering researchers with industry experience recruited by the faculty. We mainly adjusted conditions possibly influencing the eye-tracking experiment, e. g., changing a wheeled chair to an office chair without wheels or assuring stable light conditions.

D. Software Systems

To compare the cognition of microservices and monolith systems, we used open-source extensions of the Common Component Modeling Example (CoCoME), a trading system for supermarkets handling sales and stocking process [33]. CoCoME is a widely used platform for collaborative empirical research on information system evolution [34]. It is recently

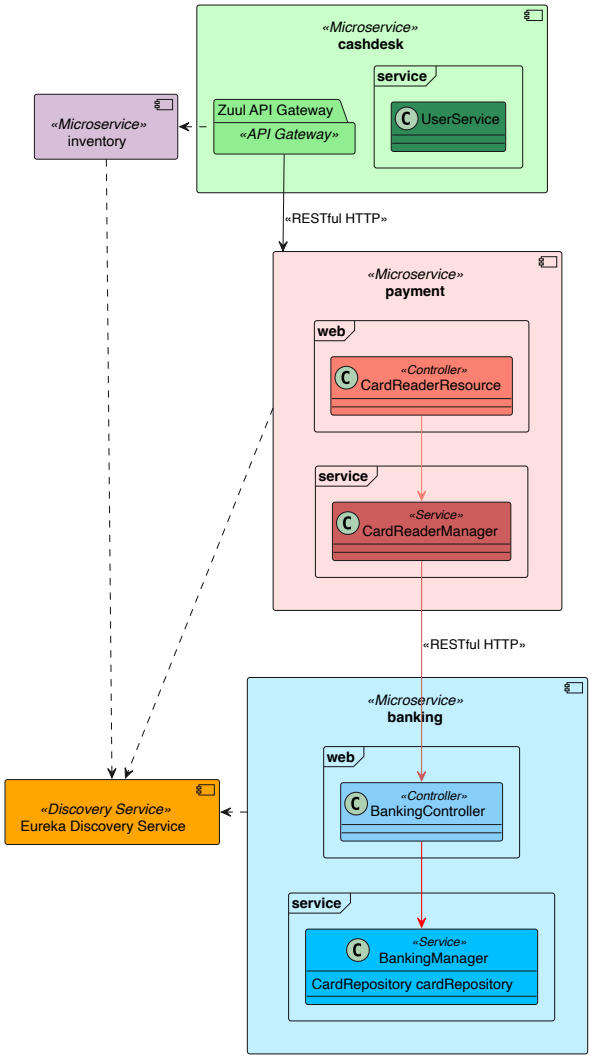

Figure 2. Excerpt of a participants' *code context model* during the *feature location* task using the **microservice** extension of CoCoME.

applied, for example, to evaluate graph-based approaches to identify microservices candidates [35].

For our study, we have applied CoCoME to a monolith[2] and decomposed it to microservices[3]. The monolithic variant (representing the control group) consists of five main components: The web application *cash-desk* frontend accesses the controllers of *Inventory*, the *BarcodeScanner* and the *CardReader* using the *Bank* service to verify debit card payments. Figure 3 shows an excerpt of the code context model a participants builds during comprehension tasks. The monolithic is implemented in Java (8 KLOCs) using *Spring Boot* and TypeScript (*Angular*). The core functionalities are located in the *services* package, i.e. the verification of the debit card in class `BankService` and the modification of a data protection-related method to delete non-active users residing in the `UserService` class.

[2]https://github.com/eveline-nuta/sales-app
[3]https://github.com/eveline-nuta/microservice-sales-app

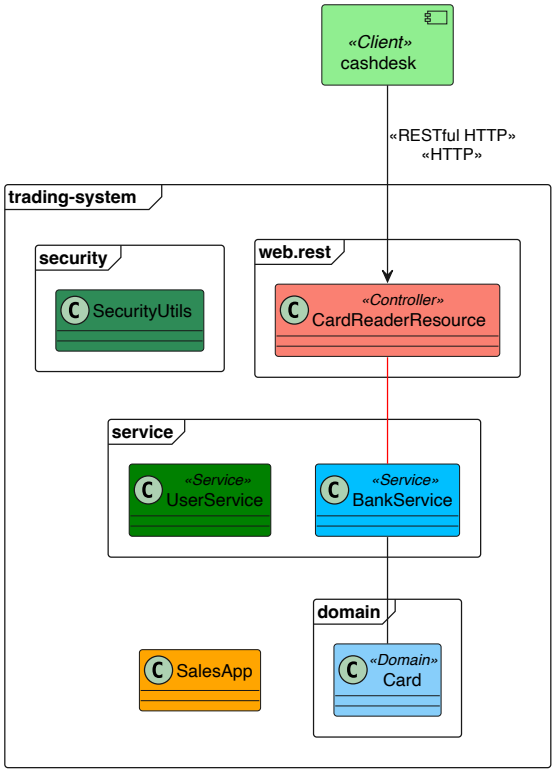

Figure 3. Excerpt of a participants' *code context model* during the *feature location* task using the **monolith** extension CoCoME.

The monolith is migrated to five microservices, the *cash-desk*, *inventory*, *payment*, *banking* microservices and a *service-registry* (see Figure 2) using *Netflix Eureka* for service discovery, *Zuul* as an API gateway and *Spring Cloud Config*. Compared to the monolithic variant, the code size differs significantly, containing more than 20 KLOC of Java code, including documentation.

E. Tasks

An essential goal of this study is to assess cognitive processes of how developers understand relationships between components, locate features and maintain real-world systems and possibly contribute to a contemporary program comprehension model.

1) Feature Location: One of the most common tasks in programming is finding the source code in a system that implements a specific feature. Due to the lack of support for automatic feature location [36], feature identification and location is still one of the most time-consuming tasks involving high cognitive effort. The prior discussed cognition models and their automated assessment could lay the foundation for future feature identification and location automation. For the *Feature Location* task assessing navigation and comprehension strategies, we asked participants the following question: "*What is the primary class to check the validity of the card?*".

2) Modification: The *Modification* task aims to study memory and problem-solving in software maintenance tasks (see Section I).

```
267 /**
268  * Not activated users should be automatically deleted after 3 days.
269  * <p>
270  * This is scheduled to get fired everyday, at 01:00 (am).
271  */
272 @Scheduled(cron = "0 0 1 * * ?")
273 public void removeNotActivatedUsers() {
274     userRepository
275         .findAllByActivatedIsFalseAndCreatedDateBefore(Instant.now().minus(3,
                ChronoUnit.DAYS))
276         .forEach(user -> {
277             log.debug("Deleting not activated user {}", user.getLogin());
278             userRepository.delete(user);
279             this.clearUserCaches(user);
280         });
281 }
```

Listing 1. Scheduler to remove not activated Users (identical in treatment and control group)

This task also involves feature location and comprehension of the implementation of the *Scheduler* pattern, enabling systems to execute jobs on a periodic or recurrent basis. Participants were asked to find and modify the *removal* of *user-related data* and perform the following code change: "*There is a functionality to delete not activated users after a certain time. Please, modify the code so that not activated users are deleted automatically after 5 days.*" The relevant method (see Listing 1) is identical in both experimental groups located at the bottom of the `UserService` class (starting at line 267).

F. Exclusion criteria

Eye-tracking research exposes many risks and causes influencing data quality or loss. Hence, we defined criteria for excluding certain recordings from further processing, particularly when the pupil position can't be reliably calculated, and set a confidence threshold of 0.8.

G. Analysis

In this study, we perform a scan path analysis, which is particularly useful to identify and analyse participants' viewing strategies to explore stimuli and solve tasks [9]. Scan paths are a series of fixations through saccades on different parts of the stimulus. We use a dispersion-based fixation algorithm detecting fixations if changes in gaze position across recorded samples are less than 1°of visual angle and, when combined, have a minimum duration of 100 ms.

To map these fixations to source code elements, we performed three steps (see Figure 4). First, we post-processed the screen recording of the IDE session for each participant using an optical character recognition engine[4] (OCR), locating areas of the IDE, e.g., the navigation and the editing window. As developers frequently scroll and resize windows, the *extractor* function processes each frame containing a fixation. Relevant source code entities are extracted using the resulting visual locations and the source code. In the third step, we partition the

[4]https://tesseract-ocr.github.io

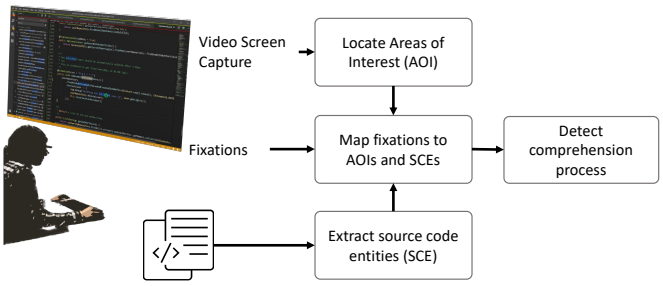

Figure 4. Post-processing steps

extracted source code entities (SCE) into chunks of logically related source code elements, spanning two and ten lines of code. The SCEs are labelled, e. g., with identifier name, classpath and identified by a sequential number j of the SCE in the program source code.

To analyse the eye-gaze behaviour, we used well-known metrics [9, 37]. The *Average Duration of Relevant Fixations* (ADRF) is the sum of the total duration of the fixations for relevant AOIs. ADRF indicates the visual effort spent on a target area (AOI) in relation to all visited AOIs, and *Time to First Fixation on Target AOI*.

For the scan path analysis, we distinguish between *reading thoroughly vs. skimming*, *scanning disorderly vs. sequential*.We calculate the saccadic length and the direction of the saccade, expressed in Euclidian distance and direction, respectively, and take scroll events into account based on the line number of code the developer fixated. The gaze behaviour is then aggregated into sets of fixations F_i and saccades S *within an SCE* and *between SCEs*.

The $cognitionscore = \frac{direction+jump}{fixationduration}$ indicates top-down and bottom-up comprehension, where the $direction$ is the Euclidian distance $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ indicating e. g., *horizontal* or *vertical* gaze movements, the *fixation duration* $> 1000ms$ indicates *reading thoroughly* and else *skimming*. *Jumps* are determined as $|j_i - j_{i-1}|$ within or between SCE_j, where $jump \leq 1$ indicate *sequential* and $jump > 1$ disorderly gaze movements. The *cognition score* is represented in the upper bars of the scarf plot (Figure 5). To calculate the regression, i.e., the revisitation of SCEs, we use the algorithm from [38] visualised in the lower bars of the scarf plot. Each SCE is colour-coded to visualise the scan path.

V. Results

1) Task performance: Table I shows the statical results, correctness and efficiency (duration to complete the task) of the 42 participants performing the *feature location* and source code *modification* task. While the *feature location* task is similar in accuracy, a significant difference can be found in the *modification* task, revealing about 8% better results for the monolith system. This is interesting as the target class (*UserService*) is identical in both systems. The time to complete the tasks is significantly higher for the microservices-based variant. For the *feature location* task, we evaluated the correctness on a more relaxed scale (see Figure 6).

Figure 7 shows the apparent trend (indicated by the red line) that more time spent working with a monolith leads to less accurate results. At the same time, subjects in the microservices group are slightly more effective when reading the code longer (Figure 6). While this partially answers **RQ1**, the comprehensibility can be measured in detail using eye-tracking recordings.

2) Scan path analysis: Table III shows the scan paths with cognition and regression scores of eight participants, exemplifying emerging patterns of all participants unsuccessfully completing the feature location task (top row) and participants completing this task successfully in the bottom row.

a) Monolith group: P15 seeks for cues related to the *card validation* feature in various security-related classes, e. g., *SecurityUtils* and fails to find the feature. *P17* also suspects the feature in security-related source code but soon changes to the retrieval-based pattern, focussing on the *card* keyword, finally arriving at the controller resource. *P18* explores the *Maven* file of the monolith, skims unit tests and security-related files, domain class, and controller (visiting 65 SCEs in total) before arriving at the target class. After reading the documentation, *P16* identifies the controller class applying the bottom-up comprehension strategy and follows the call hierarchy to the *BankService*.

b) Microservices group: P9 primarily reads (bottom-up) code related to the microservice's gateway (*Zuul*) and service registry and fails to arrive at the target class. This participant also used IDE search features, retrieving only cues in the documentation and changing back to the exploration-based pattern. *P12* visits 28 unique SCEs in different microservices and does not locate the feature but finds the related controller accessing the required functionality. *P14* completes the feature location and modification tasks in a short time (four minutes each), starting with the exploration of the file structure, skimming source code of the *inventory* microservices, returning to the file explorer, hypothesising the relevant keywords in the *UserService* class and finally comprehending the target class (*BankingManager*). In contrast, *P10* explores 211 unique SCEs, including the service registry and the declarative web service client *Feign*, conceiving the overall architecture and call hierarchies, finally leading to the target class.

3) Hypothesis testing: Table II goes into more detail and represents the statistical analysis and hypothesis testing. We use a robust non-parametric method and apply *Cliff's* δ for testing hyphotheses [39]. We hypothesise that there is no difference in comprehension and maintenance between microservices (treatment group) and monoliths (control group) and specify the following null hypotheses:

H_{0_c}: There is no significant difference between the microservice and the monolith variant in terms of time to first fixation on target AOI and the *Average Duration of Relevant Fixation* on target AOIs in relation to all visited AOIs.

H_{0_m}: The cognitive effort to maintain (modify) the microservices and the monolith variant is not significantly different.

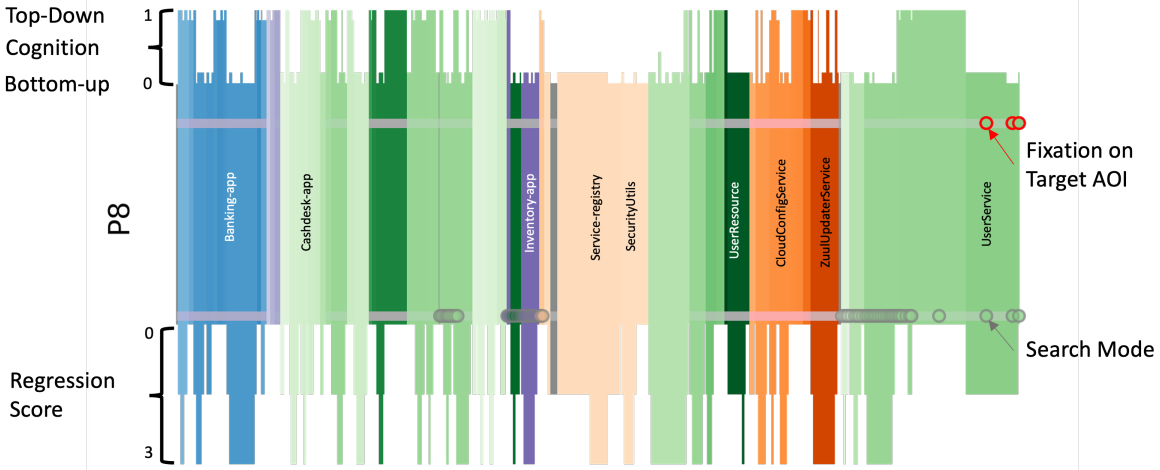

Figure 5. Scarf plot of P8 performing Modification task

Table I
PERFORMANCE RESULTS

System	Correct Feature Location	Modification	Incorrect Feature Location	Modification	Mean efficiency Feature Location (sec)	Modification (sec)
Monolith	15	20	8	3	269.17 (∓184.54)	567.73 (± 434.14)
Microservice	13	15	6	4	342.26 (∓212.80)	679.63 (± 419.42)
Total	28	35	14	7	302.24 (± 198.78)	618.35 (± 426.08)

Table II
FEATURE LOCATION COMPREHENSION PATTERNS AND METRICS

	Feature Location Task Microservices mean [ms]	Monolith mean [ms]	Cliff's δ	Modification Task Microservices mean [ms]	Monolith mean [ms]	Cliff's δ
Sum of Dwell time	212840.7	214757.9	-0.06	379470.1	343664.6	-0.12
Time of First Fixation on Target	2834761.65	2487532.5	**-0,58*** **	5824928.3	4842338.5	-0.1
Average Duration of Relevant Fixation	0.3	0.1	*-0.4***	0.1	0.1	0.0
Sum of Top-Down Dwell Time	18695.9	32101.8	*-0.38***	45142.2	59145.1	*-0.38***
Sum of Bottom-Up Dwell Time	59927.0	88321.1	-0.32*	129430.4	176262.2	*-0.4***
Sum of Dwell Time in Navigation Window	134217.7	94335.0	0.2*	204897.6	108257.3	0.22*
Sum of Dwell Time for Search	28211.7	12720.0	-0.06	42948.2	30378.3	0.08

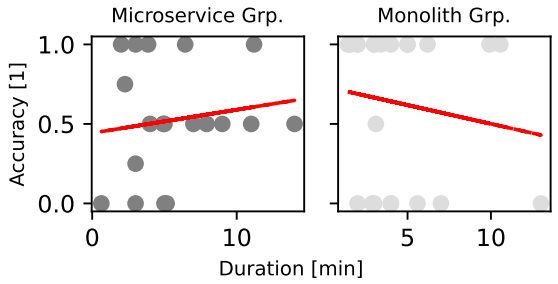

Figure 6. Feature Location Task: Scatter plot per group of dependent variables *Accuracy* to *Duration*

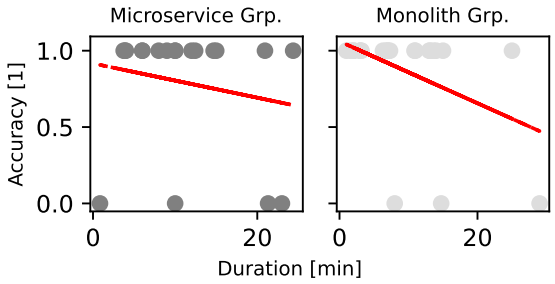

Figure 7. Modification Task: Scatter plot per group of dependent variables *Accuracy* to *Duration*

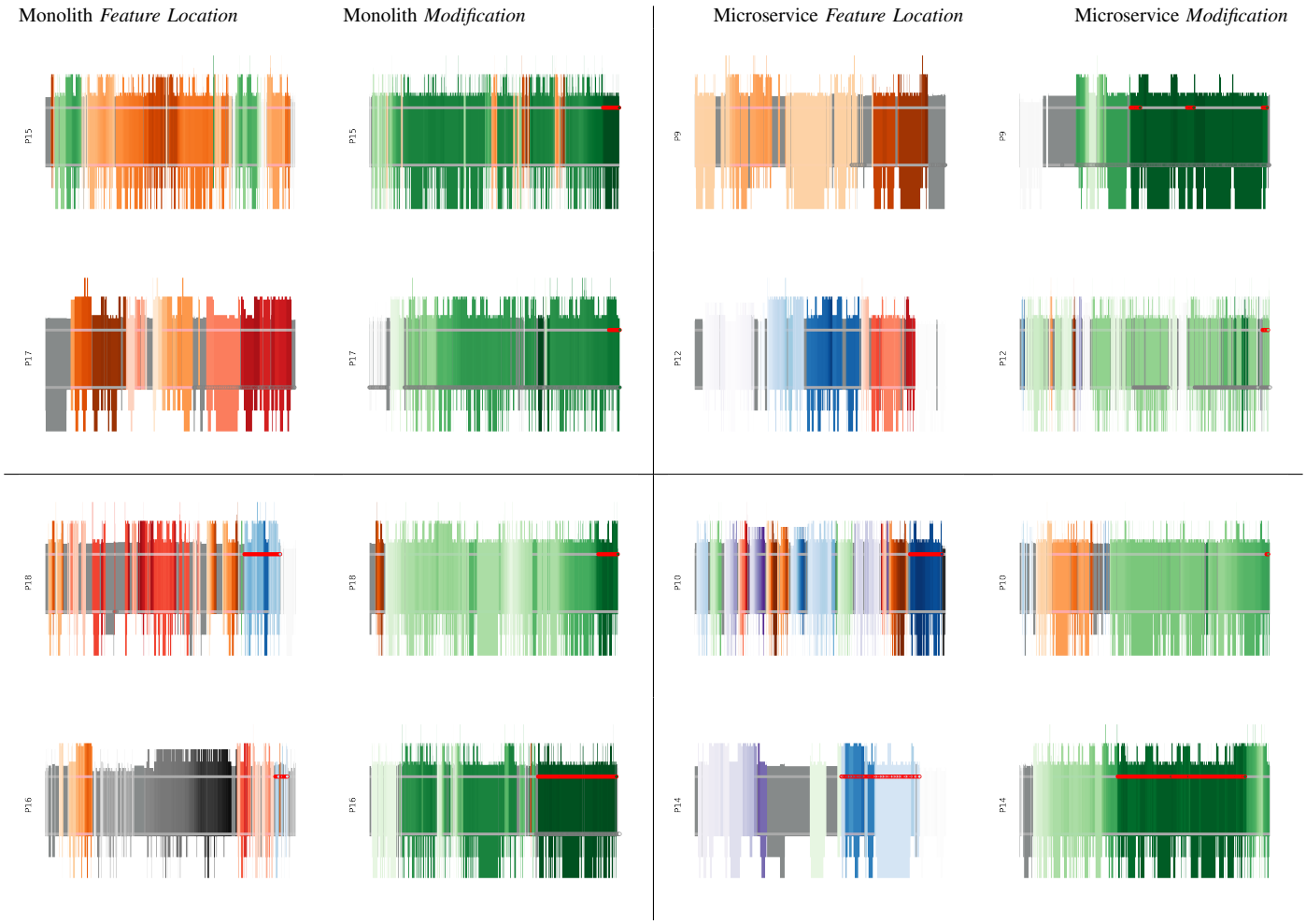

Table III
SCARF PLOTS WITH COGNITION SCORE (TOP), LOWER VALUES INDICATING BOTTOM-UP COMPREHENSION, HIGHER VALUES TOP-DOWN COMPREHENSION. REGRESSION SCORE (BOTTOM) SHOWING REVISITATION OF SECTIONS IN CODE. RED POINTS ON THE HORIZONTAL UPPER AXIS INDICATE FIXATIONS ON THE TARGET AREA, AND THE LOWER AXIS INDICATES THE IDE SEARCH MODE.

Corresponding to **RQ1**, we reject H_{0_c} as effect size, i. e., Cliff's δ is large (*-0.58*), and there is a medium effect regarding the ADRF (δ *-0.4*). We also observe a small to medium effect size of the *top-down comprehension* and the *bottom-up comprehension* strategies (δ = *-0.38* and *-0.32* correspondingly), but no significant differences in exploration-based (*dwell time spent on Navigation Window*) or retrieval-based *search patterns* patterns. Regarding **RQ1**, we conclude that participants working on the monolith variant comprehend *call dependencies* faster and spend less cognitive effort comprehending *relationships between components*.

To answer **RQ2**, we also applied Cliff's δ method and can not reject the first part of H_{0_m}. There is no significant difference between the microservice-based and the monolithic variant regarding the visual effort (*Time to First Fixation on Target area* δ=0.1). Again, there is a medium effect regarding the *top-down comprehension* and the *bottom-up comprehension* strategies (δ=*-0.38* and *-0.4* correspondingly), which is consistent with the prior result (RQ1). Regarding **RQ2**, we conclude there is no significant difference when applying conventional eye-tracking metrics, but there is a medium effect regarding the comprehension strategy. Participants comprehending microservices apply more often *bottom-up comprehension* than *top-down comprehension* strategies.

VI. DISCUSSION

This study presents the first attempt to compare cognition of architectural styles, here monoliths and microservices using an example system widely used in empirical software engineering research as the stimuli. There is no correlation between software engineering experience and task performance. Considering the relatively low difficulty level of the tasks, the completion time is relatively high, illustrating the central problem in software maintenance. This study confirms prior assumptions regarding the cognitive effort in comprehending microservices. While there is no significant difference in dwell time (duration of fixations on AOIs), developers need more time and cognitive effort to understand the relationship of

components and call hierarchies in microservices. Participants apply bottom-up strategies to comprehend various microservice patterns, e. g., *API gateway* or *service discovery* and *declarative HTTP API clients* consuming REST API endpoints exposed by microservices. This also applies to the modification task, also indicated by a higher regression rate, leading to higher visual efforts while exploring many source code entities. This result can also indicate higher maintenance costs of microservices. The popular microservice architectural style should be carefully considered, especially for small to medium-sized applications.

The study also reveals a successful search pattern for navigating microservices. Participants exploring the file tree in the seed phase are subsequently more effective. The structural cohesion of microservices helps to locate features faster for comparatively easy-to-find locations. Independent of the architectural style, participants using the IDE's search function are overwhelmed with the many results. The study's result can serve as a starting point for designing and developing more cognitively grounded tools for future microservices applications. This may include the automatic extraction of *cognitive maps* and IDE extensions for *cognitive-aware feature location*.

1) Cognitive mapping: A visual representation of the developer's mental model could assist developers in spatially mapping cognitive landmarks and relationships between components to 1) efficiently retrieve relevant seed information and entrance points to the system and 2) offload the working memory, enabling them to stay focused and find guidance when losing track of program comprehension. Research in navigational support, e.g. NavTracks [40], could use cognition models to identify source code entities relevant to the software engineer. Another research direction is the automated construction of cognitive maps using sensory inputs like gaze behaviour.

2) Cognitive-aware feature location: Our results indicate that the search functions of IDEs may even be counterproductive when frequently used keywords such as *validate* or *delete* are used. Similar to the *modification* task where the related code is located around line 280, developers need to forage large chunks of source code elements, which takes considerable time. Semantic search tools and more research are needed to support developers in constantly growing systems [36].

VII. Threats to validity

Using eye-tracking comes with intrinsic limitations [9]. We mitigated these limitations by adhering to guidelines and applying best practices. Regarding the construct validity concerning the relation between theory and observation, participants' motivation drives task performance in multiple ways. A developer who is highly interested in software artefacts can take more time to complete the task. On the other hand, less motivated developers may seem more efficient, but at the cost of effectivity [41]. Distraction in office environments, e.g., phone calls, and communication with teammates, also significantly affects attention. In our experiment, we tried to reduce any source of distraction. Results may vary in realistic work settings. To reduce the Hawthorne effect, we reduced the contact with participants to a minimum. We admit threats to external validity concern the generalizability of our findings. The experimental stimulus used (CoCoME) case study is not representative of all applications. Still, as it is widely used in empirical software engineering research, we argue that it might be indicative for many small to medium applications that do not face thousands of concurrent users. Due to the lack of a commonly available data set containing the implementations of the same system in the two different architectural styles, we decided to use the monolithic and microservices extensions of a trading system as stimuli. Furthermore, the recruited participants are representative of developers early in their careers. Approaching senior professionals would increase generalizability and improve the understanding of how architectural styles affect experienced developers with more prior knowledge. All participants are students in the final year of their studies. Referring to Kitchenham et al. [42] "...using students as participants is not a major issue as long as you are interested in evaluating a technique by novice or non-expert software engineers. Students are the next generation of software professionals so are relatively close to the population of interest". Finally, microservices are polyglot by definition. Larger studies involving more participants and code written in languages other than Java should be conducted to corroborate or contradict our results.

VIII. Conclusion and future work

In this study, we investigate how software engineers create mental representations of software applications through the lens of two different architectural styles: microservices and monoliths. Specifically, we obtain an understanding of how developers comprehend and maintain microservices versus monoliths. We present the design of a controlled experiment (n=42) using a combination of eye gaze and user interaction data and report on the analysis using a novel scarf plot technique combining cognition and memory models. We identify patterns in comprehension and provide a method to inspect top-down and bottom-up comprehension along regression visually. We performed a statistical analysis on comprehension strategies applied and applied a robust non-parametric method for hypothesis testing. Finally, we provide implications and future research directions, such as the cognitive-aware feature identification and location and a cognitive mapping approach having the potential of a paradigm shift in software architecture research as an answer to current challenges with the growing complexity of software systems. We measured differences in comprehension strategies (top-down and bottom-up comprehension) and conclude that 1) there is a significant difference in terms of visual effort and time taken to identify a specific feature, and 2) bottom-up comprehension is more frequently applied in microservices than in the monolith variant. For our future work, we plan to use this method to assess the developer's cognition of architectural designs and deployment-related artefacts and generate cognitive maps and predictive models to support software engineers making data-driven architectural design decisions.

ACKNOWLEDGMENT

This work was supported by: FWF (Austrian Science Fund) project API-ACE: I 4268.

REFERENCES

[1] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science-Research and Development*, vol. 32, pp. 301–310, 2017.

[2] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[3] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.

[4] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106600, 2021.

[5] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 286–28 610.

[6] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 13–20.

[7] J. E. Kragel and J. L. Voss, "Looking for the neural basis of memory," *Trends in cognitive sciences*, vol. 26, no. 1, pp. 53–65, 2022.

[8] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," *Guide to advanced empirical software engineering*, pp. 201–228, 2008.

[9] Z. Sharafi, B. Sharif, Y.-G. Guéhéneuc, A. Begel, R. Bednarik, and M. Crosby, "A practical guide on conducting eye tracking studies in software engineering," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3128–3174, Sep 2020.

[10] K. Holmqvist, S. L. Örbom, I. T. Hooge, D. C. Niehorster, R. G. Alexander, R. Andersson, J. S. Benjamins, P. Blignaut, A.-M. Brouwer, L. L. Chuang *et al.*, "Eye tracking: empirical foundations for a minimal reporting guideline," *Behavior research methods*, vol. 55, no. 1, pp. 364–416, 2023.

[11] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992.

[12] A. Sharma, M. Kumar, and S. Agarwal, "A complete survey on software architectural styles and patterns," *Procedia Computer Science*, vol. 70, pp. 16–28, 2015.

[13] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conf.* IEEE, 2015, pp. 583–590.

[14] H. M. Ayas, P. Leitner, and R. Hebig, "Facing the giant: A grounded theory study of decision-making in microservices migrations," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–11.

[15] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 492–501.

[16] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[17] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE TSE*, vol. 44, no. 10, pp. 951–976, 2017.

[18] L. Bidlake, E. Aubanel, and D. Voyer, "Systematic literature review of empirical studies on mental representations of programs," *Journal of Systems and Software*, vol. 165, p. 110565, 2020.

[19] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 7–18.

[20] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 202–213.

[21] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 255–265.

[22] N. Peitek, J. Siegmund, and S. Apel, "What drives the reading order of programmers? an eye tracking study," in *Proc. of the 28th International Conference on Program Comprehension*, 2020, pp. 342–353.

[23] P. Rodeghero and C. McMillan, "An empirical study on the patterns of eye movement during summarization tasks," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[24] N. J. Abid, J. I. Maletic, and B. Sharif, "Using developer eye movements to externalize the mental model used in code summarization tasks," in *Proc. of the 11th ACM Symposium on Eye Tracking Research & Applications*, 2019, pp. 1–9.

[25] Y.-G. Guéhéneuc, "Taupe: towards understanding program comprehension," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, 2006, pp. 1–es.

[26] S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra, "Impact of the visitor pattern on program comprehension and maintenance," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 69–78.

[27] B. Sharif and J. I. Maletic, "An eye tracking study on the effects of layout in understanding the role of design patterns," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[28] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 213–222.

[29] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, "Eyes on code: A study on developers' code navigation strategies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1692–1704, 2020.

[30] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[31] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 2005, pp. 181–191.

[32] R. Haas and B. Hummel, "Deriving extract method refactoring suggestions for long methods," in *Proc. of the 8th International Conference Software Quality*, 2016, pp. 144–155.

[33] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, *The Common Component Modeling Example: Comparing Software Component Models*. Springer, 2008.

[34] R. Heinrich, K. Rostami, and R. Reussner, *The CoCoME platform for collaborative empirical research on information system evolution*. KIT, 2016.

[35] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *Dependable Software Engineering. Theories, Tools, and Applications: 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings 4*. Springer, 2018, pp. 50–65.

[36] L. Di Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, 2023.

[37] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 96–103.

[38] C.-K. Yang and C. Wacharamanotham, "Alpscarf: Augmenting scarf plots for exploring temporal gaze patterns," in *CHI conference on human factors in computing systems*, 2018, pp. 1–6.

[39] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, pp. 579–630, 2017.

[40] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: supporting navigation in software maintenance," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 325–334.

[41] D. G. Feitelson, "Considerations and pitfalls in controlled experiments on code comprehension," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 106–117.

[42] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.