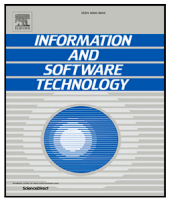
Information and Software Technology 150 (2022) 106989

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Trace visualization within the Software City metaphor: Controlled experiments on program comprehension

Veronika Dashuber [a,*], Michael Philippsen [b]

[a] QAware GmbH, Aschauer Str. 32, 81549 Munich, Germany
[b] Programming Systems Group, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058 Erlangen, Germany

ARTICLE INFO

Keywords:
Trace visualization
Software city
Program comprehension
Aggregation
Heatmap
Root cause analysis

ABSTRACT

Context: Especially with the rise of microservice architectures, software is hard to understand when just the static dependencies are known. The actual call paths and the dynamic behavior of the application are hidden behind network communication. To comprehend what is going on in the software the vast amount of runtime data (traces) needs to be reduced and visualized.

Objective: This work explores more effective visualizations to support program comprehension based on runtime data. The pure DynaCity visualization supports understanding normal behavior, while DynaCity$_{rc}$ supports the comprehension of faulty behavior.

Method: DynaCity uses the city metaphor for visualization. Its novel trace visualization displays dynamic dependencies as arcs atop the city. To reduce the number of traces, DynaCity aggregates all requests between the same two components into one arc whose brightness reflects both the number and the total duration of the requests. DynaCity also encodes dynamic trace data in a heatmap that it uses to light up the building: the brighter a building is, the more active it is, i.e., the more and the longer the requests are that it receives and/or spawns. An additional color scheme reflects any error/status codes among the aggregated traces. In a controlled experiment, we compare our approach with a traditional trace visualization built into the same Software City but showing all dependencies (without aggregation) as individual arcs and also disabling the heatmap. We also report on a second study that evaluates if an error-based coloring of only the arcs is sufficient or if the buildings should also be colored. We call this extension DynaCity$_{rc}$ as it is meant to support *r*oot *c*ause analyses. The source code and the raw data of the quantitative evaluations are available from https://github.com/qaware/dynacity.

Results: We show quantitatively that a group of professional software developers who participated in a controlled experiment solve typical software comprehension tasks more correctly (11.7%) and also saved 5.83% of the total allotted time with the help of DynaCity and that they prefer it over the more traditional dynamic trace visualization. The color scheme based on HTTP error codes in DynaCity$_{rc}$ supports developers when performing root cause analyses, as the median of them stated that the visualization helped them *much* in solving the tasks. The evaluation also shows that subjects using DynaCity$_{rc}$ with colored arcs and buildings find the responsible component 26.2% and the underlying root cause 33.3% more correctly than the group with just colored arcs. They also ranked it 40% more helpful to color both.

Conclusion: The DynaCity visualization helps professional software engineers to understand the dynamic behavior of a software system better and faster. The color encoding of error codes in DynaCity$_{rc}$ also helps them with root cause analyses.

1. Introduction

Modern software is often built according to a microservice architecture so that it can run as a distributed system in the cloud and thus benefits from the elasticity and scalability of this technology. Since a microservice is independent and focuses on a specific use case, its source code becomes smaller, less complex, and easier to maintain and comprehend than a monolith. However, the complexity is still present; by distributing the system, the complexity only shifts to the mostly asynchronous communication between components. This makes the

* Corresponding author.
E-mail addresses: veronika.dashuber@qaware.de (V. Dashuber), michael.philippsen@fau.de (M. Philippsen).

https://doi.org/10.1016/j.infsof.2022.106989
Received 10 December 2021; Received in revised form 9 June 2022; Accepted 21 June 2022
Available online 25 June 2022

traceability of invocations and the understanding of program sequences more difficult.

For understanding how and why a distributed system behaves the way it does, the so-called three pillars of observability matter: logs, traces, and metrics [1]. Logs and metrics are mainly useful for debugging. Traces provide information about the call structure of a distributed application. As static dependencies only reveal possible call paths, dynamic dependencies are conceptually more useful because they show the actual program flow. In practice however, the vast amount of dynamic trace data that a running application generates restricts the possibility to gain insights from dynamic trace data and to exploit them for program comprehension. Hence, much effort has been made to address these issues by reducing [2] or visualizing [3–6] of execution traces.

The city metaphor is widely used to visualize software [7]. Components, such as Java classes, are mapped to buildings. Containers of components, such as Java packages, are mapped to districts. Dependencies between components are usually represented with arcs drawn between buildings. We also use this metaphor in our visualization. But the representation of dependencies as arcs atop the software city does not scale as in general representing every single of the many traces as an arc overloads the visualization. Therefore, our visualization reduces the number of arcs. We aggregate all traces between the same components into one representative. Its brightness reflects the number and the duration of the aggregated traces. The visualization is a lot less cluttered. To also show the activity of each component we use a heatmap to light up the buildings. The brighter a building is, the more active it is, i.e., the more and the longer the requests are that the building receives and/or spawns.

This article extends our work presented at the VISSOFT 2021 conference [8] and also visualizes error/status codes in DynaCity$_{rc}$. With this extension the software engineers cannot only understand the normal behavior of an application, when for example HTTP status codes like 403 are frequent and expected in its operation and do not pose a problem. But they can also visualize faulty behavior and perform root cause analyses.

The remainder of the paper is structured as follows: Section 2 covers related work, specifically addressing trace reduction techniques, trace visualizations, and heatmaps. Section 3 presents (pure) DynaCity, highlighting our aggregation algorithm for displaying only a reduced set of trace representatives as well as projecting the heatmap onto the Software City. Section 4 describes our controlled experiment to gauge the impact of DynaCity's trace visualization on program comprehension. Section 5 discusses the results as well as the threats to validity. Section 6 covers and evaluates the coloring according to error/status codes that we added in DynaCity$_{rc}$. We keep the quantitative evaluation separate from Sections 4 and 5 as the numbers came from a separate controlled experiment with a different set of participants. Finally, Section 7 concludes.

2. Related work

Trace visualizations that only show the active traces at a certain point of time (either live monitoring systems or playback systems) do not need reduction techniques. For example, the playback Software City from Waller et al. [9] for the analysis of concurrency issues visualizes single execution traces that come and go over time. The monitoring system of Fittkau et al. [3] also does not need reductions.

2.1. Trace reduction techniques

Trace visualization systems for time ranges in general need reduction techniques to avoid that many arcs clutter the view. Cornelissen et al. [10] distinguish four categories of trace reduction techniques: (1) summarization techniques apply a grouping criterion and replace traces with fewer class representations. Unselected traces without a representation are not shown. (2) Metrics-based filtering approaches have in common that there are threshold values. If the metrics value of an event is below that threshold, it is hidden and does not appear in the visualization. Example metrics are a simple stack depth limitation [11,12] or the fan-in/fan-out degree of method invocations [13–15]. (3) Language-based filtering reduces traces by removing for instance getters/setters. (4) Ad hoc approaches include for example sampling.

Our approach is a combination of (1) and (2). We aggregate arcs with common endpoints. Representatives are weighted with our metrics that reflect all the call durations and frequencies of all aggregated traces. In contrast to the purely metrics-based approaches there is no threshold, i.e., all traces get visualized, but some rare invocations may end up as arcs that are almost as dark as the background.

2.2. Trace visualizations

Even after reduction techniques have been applied there may still be too many arcs so that additional techniques are needed to clean up the visualization. Frequently used are, among others, graph representations [6,16,17], hierarchical edge bundles [5], circular bundles [18], and massive sequences [19].

Caserta et al. [20] present a Software City with 3D hierarchical edge bundles. They display a line for every trace. To avoid the clutter, they organize the lines so that along common segments the lines run in parallel, i.e., they appear as thick bundles (like cable bundles). Instead of a spaghetti style chaos the visualization is tidy and easier to read. SArF Map [21] also uses edge bundles to visualize dependencies between components. Their clustering algorithm further cleans up the visualization of bundled lines. In contrast, DynaCity does not show all traces but aggregates them into fewer representatives. That also cleans out the chaos. The comparative study in Section 4 compares edge-bundling to aggregation and gauges their effects in software comprehension tasks.

In addition to cleaning up the visualization by organizing the arcs, there are also approaches that encode information from arcs into other features of the Software City. The general idea is to use heatmaps.

In the Software City by Benomar et al. [22] there are no arcs. They only project a heatmap to areas of the city's floor, while we use a more specific mapping to the brightness of the buildings/components. The authors only report on a small case study.

Krause et al. [23] add a heatmap overlay based on trace data to their Software City. This approach is similar to DynaCity, but instead of projecting the heatmap directly onto buildings, it is overlaid as additional information. Moreover arcs are not colored at all and there is no quantitative evaluation in their paper at all.

EvoSpaces [24] is a playback system that therefore does not need reductions. In addition to the usual arcs, it encodes the incoming/outgoing calls in the colors of buildings. This is similar to our heatmap. However, our metrics also reflect the duration of calls. Also, in EvoSpaces a user must explicitly map call frequencies to colors, which is hard to do for an unknown application while trying to comprehend it. Instead of predefined colors, we automatically scale light intensities to the metrics. Furthermore, in contrast to the EvoSpaces literature, we quantitatively examine the impact of our visualization on program comprehension.

There are also many approaches of software cities based on purely static analysis [7,25–27]. We do not discuss them in detail here, since they do not analyze dynamics and are therefore not close enough to our approach and goals.

Distributed tracing systems

Distributed tracing systems like Zipkin,[1] Jaeger,[2] or HTrace[3] are widely used in the industrial context. They also provide visualizations

[1] https://zipkin.io/.
[2] https://www.jaegertracing.io/.
[3] https://htrace.org/.

for single traces, similar to the schematic illustration in Fig. 1. However, these frameworks are designed for collecting large amounts of traces and supporting debug tasks in distributed environments, but not for aggregating and showing the big picture as we do. Guo et al. [28] go one step further. They abstract traces into different paths and group them into business flows. Their visualization is graph-based and they also use coloring for marking errors or latencies. Their approach is helpful in performing problem diagnoses based on the dynamic trace data. However, they do neither take source code that is not instrumented nor the static structure of it into account. In contrast, with DynaCity both can be analyzed.

2.3. Visualizations for root cause analyses

Whereas some of the trace visualizations mentioned above use colors to encode some information in the visualization (Caserta et al. [20] use the line color to represent its direction and quantification, SArF Map [21] visualizes source and target by means of its color scheme), we could not find any software city visualization that is meant to help with root cause analyses.

This task is more commonly addressed in log or network visualizations, often with call matrices [29–31] that represent errors between components or servers as circles at the corresponding position in the matrix. In contrast, DynaCity$_{rc}$ directly shows the calls, encodes their durations and frequencies in the arcs' brightness, and also provides aggregated error codes in the colors of arcs and (optionally) buildings.

Sedlar et al. [32] use a graph to visualize network topology. Like in DynaCity$_{rc}$ they also color edges and nodes based on the average streaming error. But their publication lacks a controlled experiment on how much color is beneficial and on whether coloring is useful for root cause analyses.

The Tudumi system [33] visualizes file and server accesses in usage logs in a spherical way: it places servers on circles (with a texture) and connects them to users (as squares) with straight lines, whose colors reflect the error/success of an access. This is similar to DynaCity$_{rc}$ whereas Tudumi's circles and squares are uncolored. Again there is no quantitative evaluation on whether coloring is helpful, and if so, how much color is best.

Concluding, it is a novel idea to visualize both traces and component load by means of light intensities. The combination of both and its impact on program comprehension has not been sufficiently researched. As far as we know, no other trace visualization has been the subject of controlled experiments to gauge its benefits for program comprehension and root cause analyses, nor on the effect of the extend of error-encoding color.

3. DynaCity

With DynaCity we propose an aggregated trace visualization using light intensities in a night-view of a Software City. The use of light fits perfectly into the city metaphor: imagine an office building at night, the more employees are still working at their desks, the more lights are switched on, the brighter the building is. This metaphor can be easily transferred to software components: if a component is called frequently or has a long processing time, the building of this component in the software city also shines bright. This heatmap renders component activities apparent.

In addition to building activity levels, DynaCity also visualizes dynamic dependencies, especially traces to show who calls whom, even across the static dependencies, for example when APIs call each other via HTTP, which is not detectable in a static code analysis. We import the static dependency graph (even across multiple microservices) as well as all the collected trace data. The user can freely select the time range from that data (from a few milliseconds up to the entire imported time period) for which DynaCity should visualize the data, i.e., aggregate the traces.

DynaCity uses the layered layout we implemented in our earlier work [34]. The layout arranges buildings in layers according to their static dependencies. Buildings in the front depend on buildings in the back. Systems without cyclic dependencies, like the one we use in the evaluation, hence do not have any explicit arcs for their static dependencies. Since most dynamic dependencies follow static ones, the layered order also reduces the visual clutter of the dynamic dependency arcs, as they also in general point from the front to the back instead of crisscrossing the city. The height of a building represents the component's fan-in and the building's square area reflects the component's fan-out. Both tall and wide buildings are important for the architecture of the system being visualized.

A design company advised us to use a neon color scheme as it mimics what is popular in current computer games. The users of our visualization are software developers who often also play computer games. Since for pure DynaCity we are neither evaluating the runtime behavior nor addressing root cause analyses, here and in the upcoming two sections on the first controlled experiment and its results we use neutral colors (turquoise, magenta, and orange). In Section 6 we introduce the extension DynaCity$_{rc}$ that also uses green and red to visualize error/status codes.

3.1. Data preprocessing

DynaCity works with collected trace data. We used the Elastic stack[4] to gather trace data from an application but every other tracing framework would also have worked. The user selects a time range for which to visualize all the trace data. A trace is a complete call path through the system, see Fig. 1 for an example trace in a typical illustration. It consists of several so-called spans that represent the individual calls in a trace. Child spans (callees) are plotted below parent spans (callers), i.e., in the example the `booking` service calls the `payments` service. The width of a span corresponds to its duration. In general, a full span is within the user-selected time range [from_timestamp; to_timestamp]. Then there is no need for preprocessing of the span. If however a span/call is initiated before the from_timestamp or if a span lasts longer than the to_timestamp then the span's duration needs to be adjusted so that only the fragment of the span that is within the user-selected time range counts as its duration. In the example, the durations of the `booking` span and the `message` span need to be adjusted.

3.2. City heatmap

Heatmaps help to quickly find those areas in a large dataset that matter most. A heatmap based on the endpoints of execution traces of a software system helps to quickly find the key components that are used a lot (as callees and callers). Just as well one can seize which components are rarely or even not used at all. To stay in the metaphor that more light means more activity, we use the light intensity of a single color instead of a color scheme. To determine the brightness of a component i, we use the *component load*, a combined metrics of the call frequency and the call durations:

$$load_{comp}(i) = \sum_{k \in C_i} \frac{k}{s} \tag{1}$$

where C_i is the set of durations of all calls that reach component i or are spawned by i and s is the length of the user-selected time range.

Assume we only have the example trace in Fig. 1. Then the component load of the `cinema catalog` component equals the processing duration of the `GET /catalog/<<city id>>` call plus the waiting time for the return value of the spawned `GET /movies/premieres` call from the `movies` service. The component load of the `movies`

[4] https://www.elastic.co/elastic-stack/.

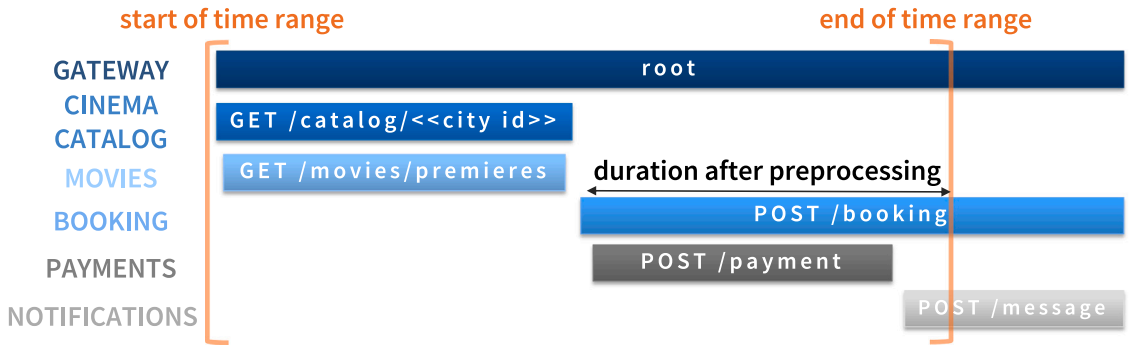


Fig. 1. Preprocessing of trace data for a user-selected time range.

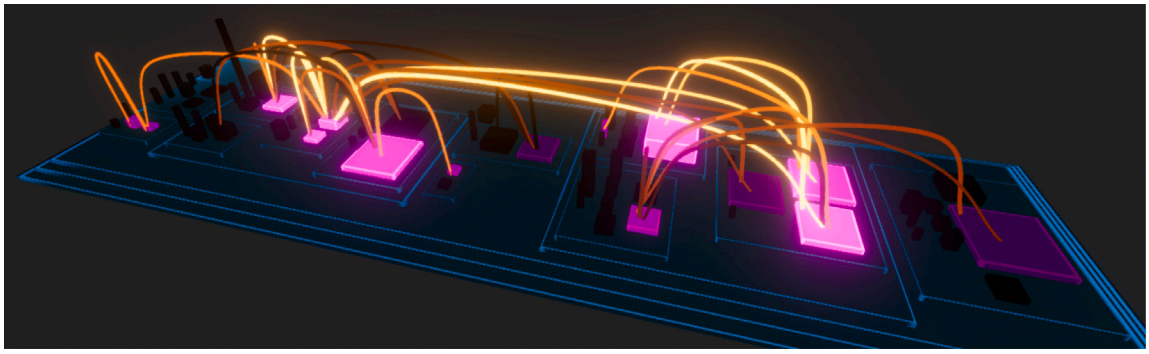


Fig. 2. DynaCity visualizes both the *component load* and the *call load* with light intensities.

service is smaller as it only equals the duration for processing the GET `/movies/ premieres` call.

To visualize component loads we have to map the cumulated values to brightness values. As there are many outliers we first use an interquartile range (IQR) normalization with 25% / 50% / 75% quartiles Q_1 / Q_2 / Q_3:

$$load'_{comp}(i) = \frac{load_{comp}(i) - Q_2}{Q_3 - Q_1} \tag{2}$$

Then we map the resulting $load'_{comp}$ linearly to an interval [min; max] of brightness values.

In Fig. 2 components with a higher load shine brighter. Components that are not active at all in the selected time range are almost dark, for example adjacent to the bright magenta building in the lower right or on the left in the back. (The height of a building reflects the component's fan-in and the square area stands for its fan-out.)

3.3. Aggregated arcs

It would clutter the visualization, if for every single trace there was an arc between its endpoints. We reduce the number of arcs by grouping together all traces with the same endpoints in one representative arc. We again use the light intensity to express the importance of such an aggregated arc by summing up all the durations of all the traces it represents. The metrics for the *load* of a *call* between components i and j is:

$$load_{call}(i,j) = \sum_{k \in C_{i,j}} \frac{k}{s} \tag{3}$$

where $C_{i,j}$ is the set of durations of all calls between i and j. For normalization to the user-selected time range we again divide by its length s.

Assume again that we only have the example trace in Fig. 1. Then the call load from the `cinema catalog` component to the `movies` component equals the processing duration of the GET `/movies/ premieres` call. If we had this call n times, the call load would also be n times the processing duration of GET `/movies/ premieres`.

Afterwards we again apply an interquartile range normalization to the *call loads* before we linearly map the resulting values $load'_{call}$ to the interval of reasonable light intensities.

The brighter the arcs shine in Fig. 2, the more the two connected components communicate with each other.

3.4. Step through feature

What has been discussed above for one user-selected time range, Dyna-City actually makes available for a series of consecutive time segments. Data is preprocessed per segment, there is one visualization per segment, and load values are normalized by dividing them by the segment size s. Users can step between segments to spot changes in call and component loads.

4. Controlled experiment on DynaCity (no color-encoding yet)

The main objective of this first study is to quantitatively evaluate the (pure) DynaCity visualization in comparison to a more traditional visualization that uses edge-bundled arcs and that does not employ a heatmap for component loads. There is no color-encoding of errors to help with root cause analyses yet — Section 6 discusses and evaluates this extension separately.

Fig. 3 visualizes the same trace data that DynaCity displays in Fig. 2. Compare for example the two aggregated arcs that start/end in the component in the lower right. In the edge-bundled variant, there are two bundles instead. In the component's package DynaCity lights up one active component, while in the traditional view all the buildings have the same brightness. All other visual properties and the layout are identical for both variants.

More formally, the study aims at answering the following three research questions:

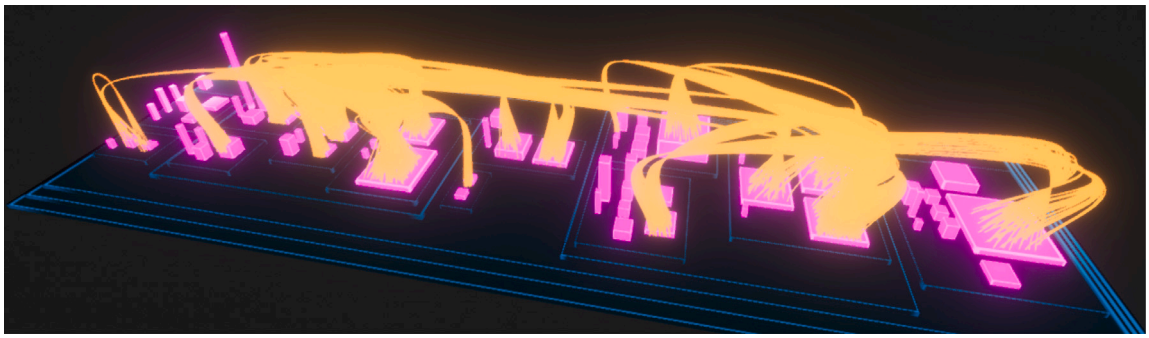


Fig. 3. Traditional edge-bundle visualization, no heatmap; same layout as in DynaCity.

1. Does the use of DynaCity reduce the time that is needed to solve software comprehension tasks?
2. Does the use of DynaCity improve the correctness of the answers given?
3. Which visualization do the software engineers prefer?

To answer the questions, we set up a first controlled experiment. The general study design was as follows: For a demo application and a set of software comprehension tasks (see Section 4.1) we measured the time that professional software engineers (see Section 4.3) spent on them and how correct their answers were. We randomly assigned the subjects to two groups, one group had to perform the experiment with the DynaCity visualization the other with the more traditional view. In addition to the visualization, the subjects also had access to the source code but none of the subjects opened any of the files. Section 4.4 sketches the experimental procedure in more detail. After the subjects worked on the tasks they were given an exit questionnaire (see Section 4.2), comprising of the NASA-TLX questionnaire and some additional questions. The subjects then also received screenshots of the respective other visualization, as well as a short description. In the additional questions the subjects had to compare the other visualization with the one they used for the study.

We analyzed the subjects' performance on the tasks and their responses to the exit questionnaire with respect to three hypotheses:

H1: The use of DynaCity reduces the time spent in solving software comprehension tasks.
H2: The use of DynaCity increases the correctness of the answers.
H3: Software engineers prefer to solve software comprehension tasks with DynaCity.

The corresponding null hypotheses of course are that there is no positive effect caused by DynaCity. The underlying rationale of H1 is that with an aggregated/reduced number of visible arcs, subjects quickly get an overview and almost instantly identify both the hotspots and the important call paths. H2 is based on the assumption that aggregation causes less clutter than edge-bundling so that subjects can better focus on the task and are less distracted. If the first two hypotheses hold, it is likely that professional software developers also prefer the DynaCity visualization (H3).

4.1. Object & tasks

We used the open source blogging project *Spring Boot Realworld Example App*[5] (with a small modification of its microservice architecture). We chose this project as our study object because it is open source, written in Java, which all our subjects are familiar with, and has a realistic size of about 44 KLOC that is deemed not to overwhelm the subjects. We generated trace data using load tests that ran for a total of about 15 min. As the length of the time segments that subjects used for stepping through the trace data should not affect the results of the study, we eliminated this variable and fixed it to one-minute intervals for all subjects and both variants.

Table 1
Comprehension tasks and coverage of the activity catalogue of Pacione et al.

Task	Description (allotted time)	Activities
T1	Understand and describe the typical user interactions with the application. (5 min)	A1, A7, A9
T2	Identify a highly active component. (2 min)	A4, A6
T3	Identify a rarely used component. (2 min)	A4, A6
T4	Give an example of an unused component. (2 min)	A4, A6
T5	Assume you want to change the interface of the Article API, the team(s) of which other component(s) should you inform about it? (2 min)	A2, A4, A5
T6	Describe the internal call structure of the user service. (5 min)	A1, A3, A8

To ensure that our study verifies the effectiveness of visualizations in a real-world context and that it is also unbiased toward either variant, our task design followed the advice by Pacione et al. [35]. They identified nine principal comprehension activities, and claim that "a set of typical software comprehension tasks should address all of these activities":

A1. Investigating the functionality of (a part of) the system
A2. Adding to or changing the system's functionality
A3. Investigating the internal structure of an artefact
A4. Investigating dependencies between artefacts
A5. Investigating runtime interactions in the system
A6. Investigating how much an artefact is used
A7. Investigating patterns in the system's execution
A8. Assessing the quality of the system's design
A9. Understanding the domain of the system

Even though the activities of Pacione et al. may not cover all tasks worth thinking about, we follow them nevertheless in favor of a methodologically correct study design. Table 1 lists our comprehension tasks and shows which of the above activities they cover. Each of the nine activities is covered by at least one task. The table also gives a time limit for each task that we determined in a preliminary study (with subjects that later did not participate).

Closed questions, for which there is no doubt if an answer is right or wrong, have the risk of subjects just guessing. On the other hand, open questions, have the disadvantage that a supervisor needs to determine a percentage of correctness and that thus the subjective evaluation may be biased. By having both closed and open questions as our tasks, we spread the risk of both types of disadvantages. The questions of tasks #1 and #6 are open. Tasks #2 to #5 are posed as closed questions, the subjects could only answer incorrectly or correctly. To reduce the risk of a biased assessment, we prepared a sample solution beforehand that

[5] https://github.com/gothinkster/spring-boot-realworld-example-app.

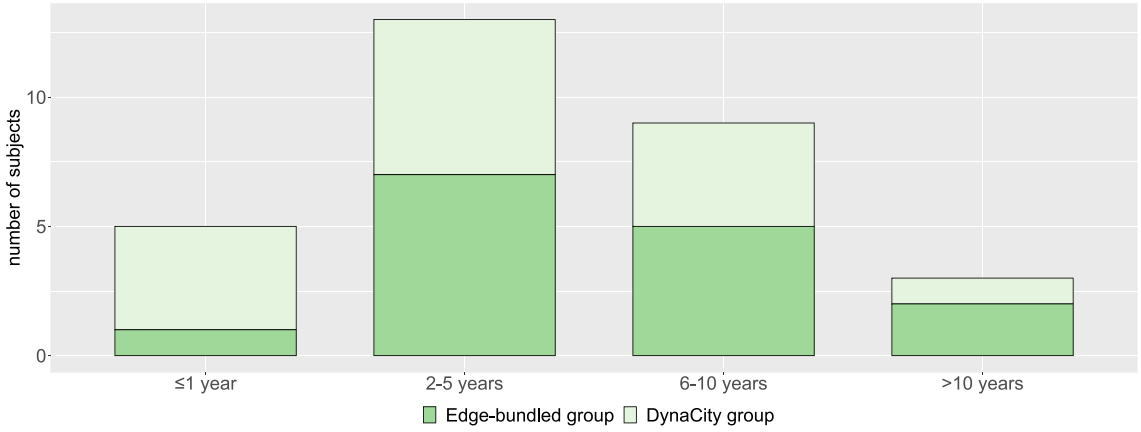


Fig. 4. Work experience. Median for both groups is 2–5 years.

listed all the aspects that should be mentioned in a perfect answer (= 100%). While evaluating the subject's answer the supervisor checked off the aspects from that list and ranked the answer in a $[0\%; 100\%]$ interval of correctness.

4.2. Exit questionnaire

In addition to working on the tasks, the subjects had to fill out an exit questionnaire consisting of two parts: the standardized NASA Task Load Index (TLX) questionnaire from Hart et al. [36] to measure the perceived workload plus additional questions directed at our third research question. Here we asked (1) how much the visualization helped the subjects to solve the tasks, (2) whether the other visualization variant would have helped them more, and (3) a comparison of the two variants in terms of clarity, comprehensibility, completeness, and personal preference. For answering (2) and (3), subjects received screenshots and a description of the other visualization together with the exit questionnaire.

4.3. Subjects

A total of 30 fully employed professional software engineers (not students) conducted our study. They all have a computer science or similar background and are familiar with dynamic dependencies and traces. They were neither familiar with the Software City visualization nor with the object (the blogging project). The professional experience of the subjects ranged from 1 month to over 10 years, while the majority had a work experience of 2–5 years (43.3%), see Fig. 4. Subjects were employed by the same IT company that gave them permission to participate in the study during normal business hours.

We randomly assigned subjects to one of two groups: 15 subjects performed the study with the DynaCity visualization, hereafter called DynaCity group, and 15 with the edge-bundled visualization of all traces, hereafter called edge-bundled group. The median work experience was 2–5 years in both groups. We had two female subjects in each group. Subjects were not aware of the purpose of the study. They neither knew the origin of the two variants nor the three hypotheses.

4.4. Experimental procedure

The experiment was performed remotely, one subject at a time and alternating between the two groups to eliminate any effects that repetition may have on the supervisor. A subject received an executable of the group's visualization (and the object's source code that was not touched) beforehand. During the study, a subject shared her/his screen and was in a video conference with the supervisor. In a short familiarization phase (about 10 min), the supervisor used a script to explain how to navigate and interact with the visualization, its visual properties, and the direction of the dynamic dependencies. A subject was allowed to ask questions.

Afterwards, the actual study started and lasted for about 20 min. The tasks were posed one after the other and the supervisor noted both time and correctness for each task. The subjects were not given any feedback on the correctness of their answers to avoid affecting the answers in the exit questionnaire. Subjects that did not finish a task well before the allotted time was up, were given a warning signal so that they were able to conclude. Upon completion of all six tasks, the video conference was ended, the exit questionnaire and the information on the other visualization variant were sent to the subjects, together with a request to work on them immediately. We did not record how long subjects worked on the exit questionnaire.

5. Results and discussion of the DynaCity controlled experiment

This section discusses our interpretation of the first study. After outlining the time measures in Section 5.1 we then describe the comprehension scores in Section 5.2. The results of the NASA-TLX questionnaire are explained in Section 5.3 and those of the additional questions in Section 5.4. Finally, we address threats to the validity of our study.

5.1. Time

We begin with testing the first null hypothesis stating that the use of DynaCity does not affect the time needed to complete typical software comprehension tasks. Fig. 5 shows the boxplot of the time measures for each task (percentage of allotted time) as well as the total time. Because of the warning signals, all subjects provided answers to all tasks within the limit. The (median of the) edge-bundled group was only faster than the DynaCity group in solving task #4. From the comments of the subjects during the study as well as from the additional remarks in the questionnaire, we know the reason for this. This task consisted of finding components that are not active at all. In DynaCity, these buildings are completely dark, which unfortunately on some screens made it very difficult to distinguish them from the background. That is why the subjects in the DynaCity group took longer to find the inactive components. In task #5, the median of the two groups is on par. Both groups took the same time to identify the other developer team that needs to be informed about an API change. The reason is that the component most relevant for this task only has few dependencies to the API component. Few arcs in the edge-bundled variant are quite similar to one aggregated arc in the DynaCity variant.

The DynaCity group was faster on the other four tasks. This confirms our assumption that the heatmap helps to quickly identify key components (task #2). DynaCity's aggregated arcs help to recognize the call

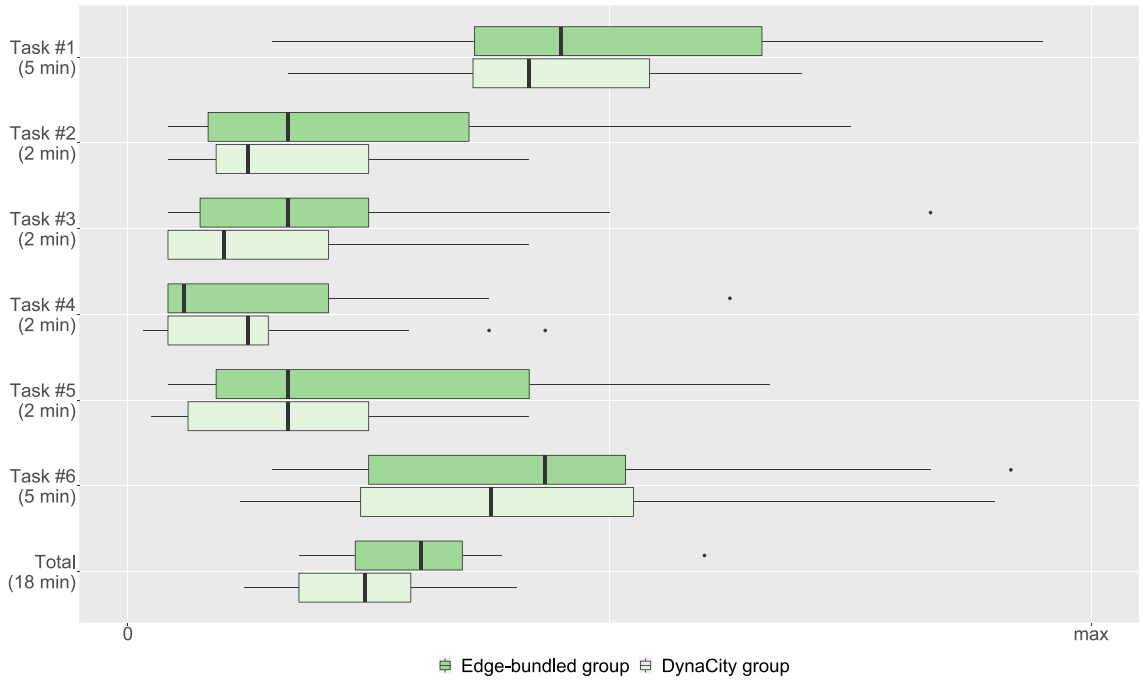


Fig. 5. Time spent on the tasks. Edge-bundled group in dark green on top and DynaCity group in light green below. Boxes correspond to the first and third quartiles (the 25th and 75th percentiles), whiskers drawn using Tukey method (1.5 IQR), points are outliers in the data. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

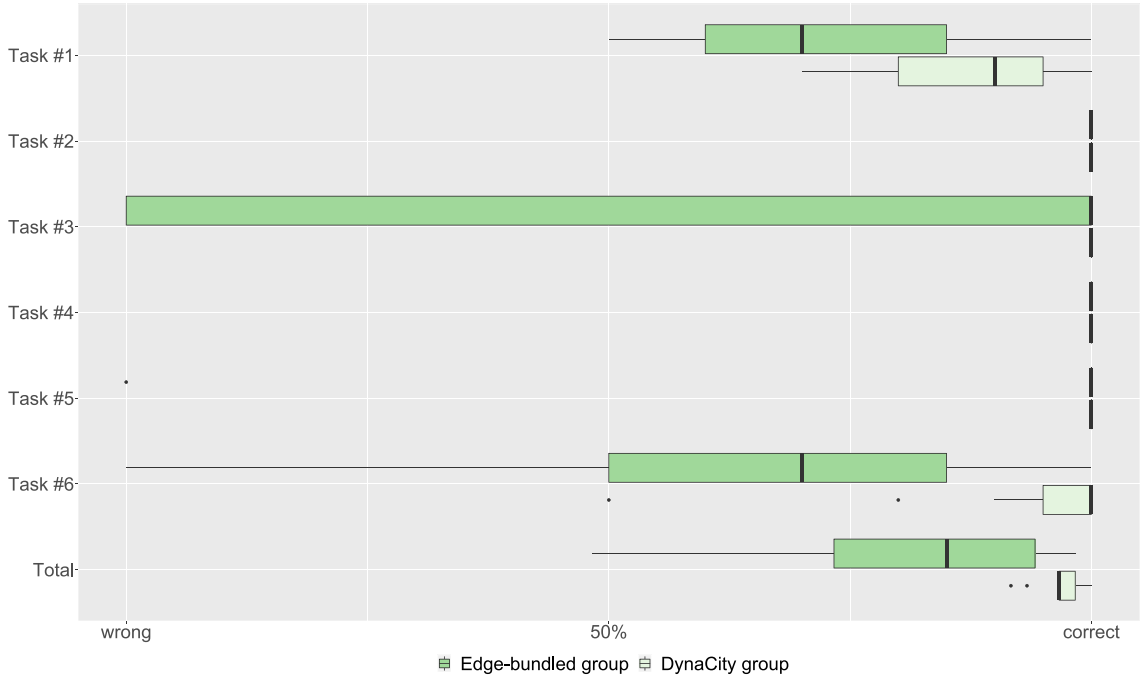


Fig. 6. Correctness of answers. Boxplots as in Fig. 5.

structure faster (tasks #1 and #6) since the reduced number of arcs makes it easier to keep an overview.

The edge-bundled group needed 5:29 min = 30.4% (median) of the total allotted time, whereas the DynaCity group needed only 4:26 min = 24.6%, thus spent 1:03 min less on solving the tasks. Hence, the DynaCity group was 19.1% faster and thus saved 5.83% of the total allotted 18 min. This is a statistically significant result (significance level $\alpha = 0.05\%$ determined by a t-test). With this finding, we can reject the null hypothesis in favor of hypothesis H1: using DynaCity reduces the time it takes to solve typical software comprehension tasks.

5.2. Comprehension

The second null hypothesis was that the use of DynaCity has no effect on the correctness of the responses. Fig. 6 shows the boxplot of the comprehension scores.

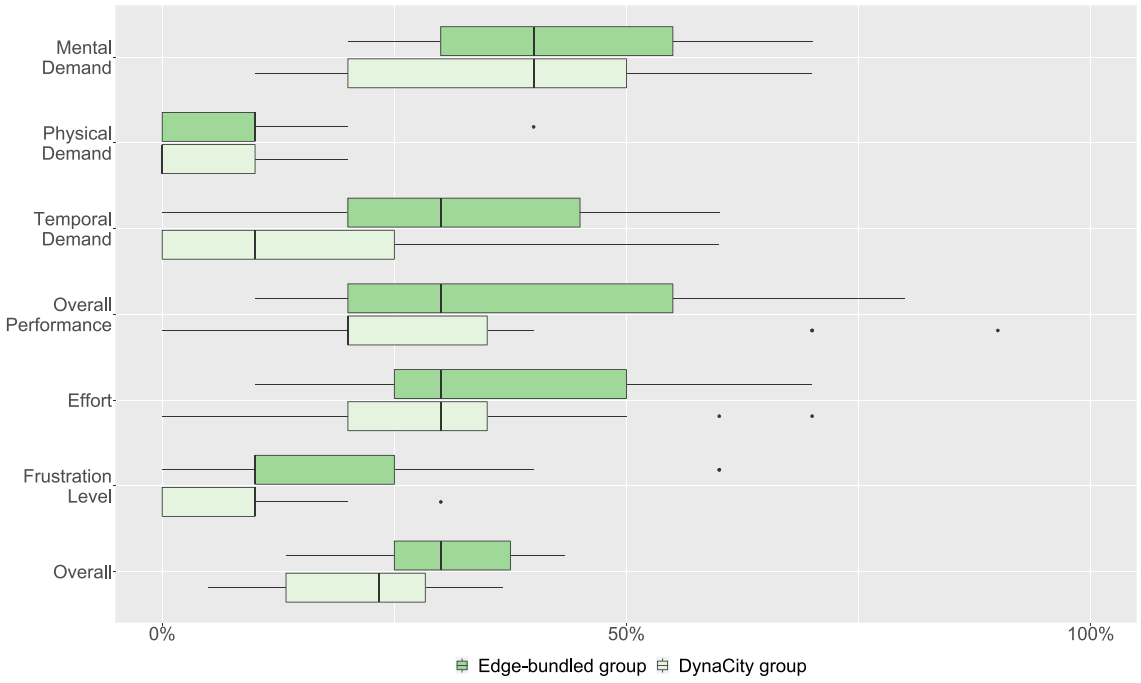


Fig. 7. Questionnaire (NASA-TLX). Evaluation of the perceived workload. 6 dimensions plus overall load. Boxplots as in Fig. 5.

For the closed questions #2 to #5 both groups were able to give correct answers (median), i.e., even with the visually cluttered edge-bundled visualization. Yet task #3 already indicates that it is more difficult to tell different dependencies apart in the edge-bundled visualization since although the median of the edge-bundled group is on the far right (correct answer), six subjects (= 40%) of this group answered the question incorrectly. In the more complex tasks #1 and #6 differences are more pronounced. According to Pacione et al. these tasks are aimed at a comprehensive understanding of the application as a whole, its quality, and internal structure, while other tasks need a more specific understanding. Especially the results of task #6 (internal call structure of a service) show that the aggregated arcs make it significantly easier for the subjects not to overlook or misinterpret the invocations. Due to the reduced number of arcs, the subjects were less distracted and were all (median) able to correctly identify 100% of the call structure.

This results in a median overall correctness of 85% (5.1 points out of 6) for the edge-bundled group compared to a statistically significantly better 96.7% (5.8 points) for the DynaCity group (11.7% improvement, significance level $\alpha = 0.05\%$ determined by a t-test). Therefore, we can also reject the null hypothesis in favor of hypothesis H2: the use of DynaCity increases the correctness of the answers.

5.3. NASA task load index (TLX) questionnaire

The NASA TLX questionnaire ranks in a standardized way the perceived workload while solving the tasks. We argue that a lower load also leads to a preference for using a visualization. Fig. 7 shows the boxplot of the six load dimensions as well as the overall task load index.

In all six dimensions and in total the workload is lower for the DynaCity group. The difference in the temporal demand (3rd dimension) is particularly striking. While the edge-bundled group rated the temporal demand as 30%, the Dyna-City group reported a statistically significant lower/better demand of 10% (median). This result supports our assumption that users find their way around faster in Dyna-City's aggregated view and thus do not consider the temporal aspect as burdening as with the traditional visualization. The assessment of the overall performance is also lower for the DynaCity group by a statistically significant 10%, indicating subjects were more confident that their answers were correct.

The median of the *overall* Task Load Index is 30% for the edge-bundled group compared to a significantly lower and better 23.3% for the DynaCity group (significance level $\alpha = 0.05\%$ determined by a t-test), i.e., a lower overall cognitive load by 6.70% (percent points). The results already suggest to reject the third null hypothesis and favor of H3, but let us also look at the answers to the additional questions.

5.4. Additional questions

In the exit questionnaire we also asked:

1. "How much did the visualization help you in solving the tasks?", scale: 0 = little, 4 = much.
2. "Would the other visualization have helped you more in solving the tasks?", scale: 0 = much less, 4 = much more.
3. "Rate the assigned visualization in comparison with the other one with respect to (a) clarity, (b) comprehensibility, (c) completeness, and (d) personal preference".

With these questions, we get comprehensive feedback from the software developers on which visualization they prefer.

Fig. 8 shows the boxplots for the first two questions. The DynaCity group rated the helpfulness of their visualization to be 90%, while the edge-bundled group only attested a significantly lower helpfulness of 70% (significance level $\alpha = 0.05\%$ determined by a t-test). The difference becomes even clearer with the second question. The second boxplot shows that while the DynaCity group is convinced that the non-aggregated view would have helped them *less*, the median of the edge-bundled group is convinced that the DynaCity visualization would have helped them *much more*.

Finally, we asked the subjects to compare the visualization they used for the study with the other one regarding four aspects: clarity, comprehensibility, completeness, and personal preference. Subjects could prefer their visualization, the other variant, or rate both as equal. Fig. 9 shows the answers that the edge-bundled group gave when comparing their variant to the aggregated DynaCity visualization. The

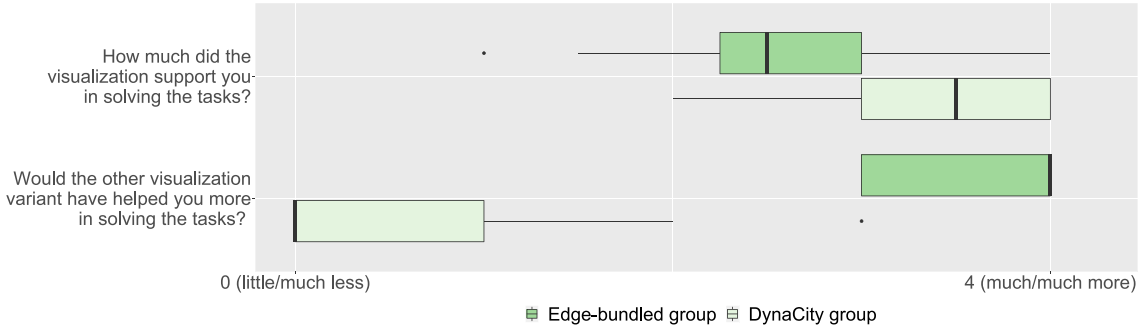


Fig. 8. Additional questions (1) + (2). Boxplots as in Fig. 5.

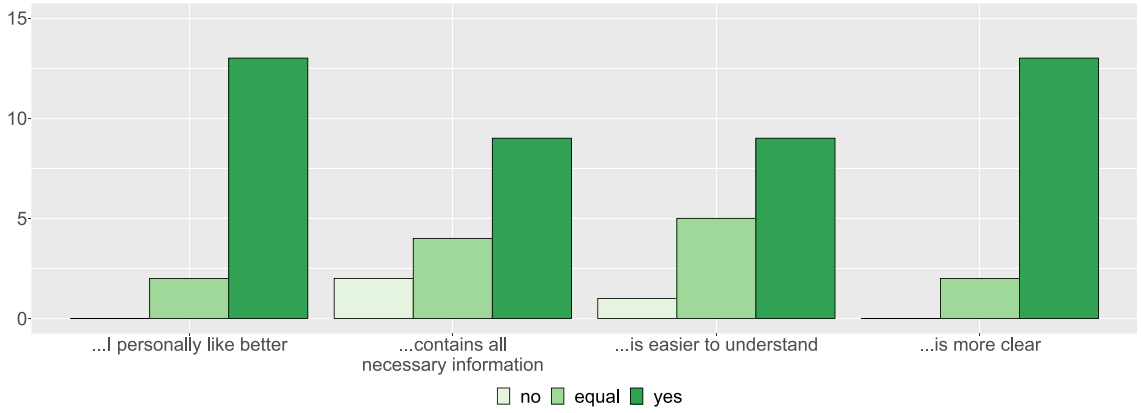


Fig. 9. What the edge-bundled group thinks about DynaCity. Read as "The Dyna-City visualization..." and attach a bar's legend.

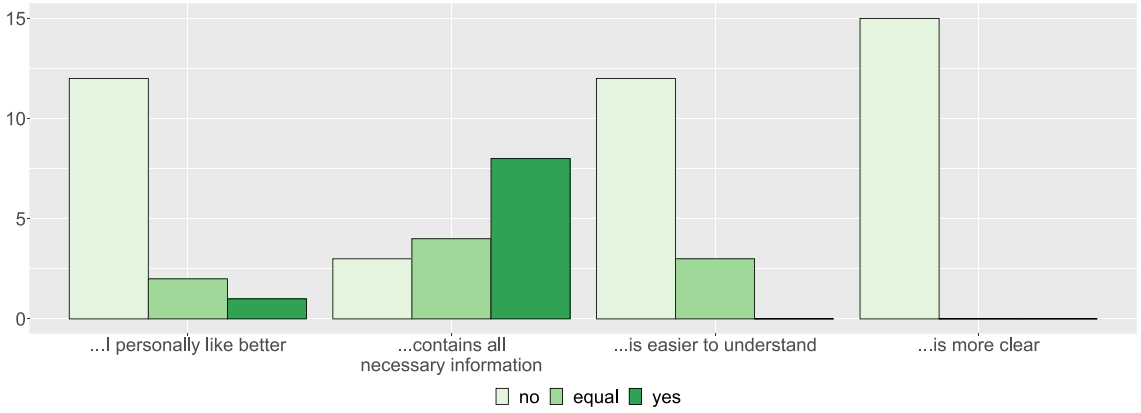


Fig. 10. What the DynaCity group thinks about the edge-bundle visualization. Read as "The edge-bundled visualization..." and attach a bar's legend.

responses given by the Dyna-City group regarding the edge-bundled visualization can be found in Fig. 10. Both groups largely agreed that the respective other visualization also provides all the necessary information (second set of bars). Since DynaCity only shows aggregated arcs instead of all traces as in the edge-bundled variant, this insight is crucial, because despite the aggregation, all information for solving the tasks is still available. For the other aspects, the preference is clearly on the side of DynaCity. The majority of the edge-bundled group is convinced that DynaCity's aggregated visualization is more clear (fourth set of bars) and easier to understand (set 3). All subjects personally liked the DynaCity visualization better (set 1). In contrast, the majority of the DynaCity group denied these attributes to the edge-bundled visualization (sets 3 and 4) and preferred DynaCity (set 1). The results are statistically significant (significance level $\alpha = 0.05\%$ determined by a t-test).

Due to the results of both the NASA-TLX and the additional questions we can reject the null hypothesis in favor of hypothesis H3: professional software engineers prefer DynaCity over the edge-bundled visualization.

5.5. Threats to validity

Let us discuss potential threats to validity along the design of the first study. We kept those threats low.

Object. Working with only one object in one programming language is a potential threat. However, none of the subjects looked into the code. They all relied on the runtime data from an instrumented application, which is language agnostic. When choosing the object, we made sure to pick one that is typical these days, i.e., with several microservices, a classic layered architecture consisting of APIs, business

logic, and database integration. Because of the typical object with common features we expect the results to be generalizable. Adding another object would have required twice as many subjects to keep the results significant.

Tasks. By choosing tasks that cover all typical software comprehension activities according to Pacione et al. we can assume that there is no bias. However, our study is about understanding and solving tasks on a formerly unknown software system. For software engineers familiar with a system and its dependencies, the measured effects on time and correctness may have been less pronounced. Still, the results of the comparison of the two visualizations regarding clarity, comprehensibility, and personal preference are generalizable to software engineers working on a known system.

Variants. Another potential threat is that we may have implemented the edge-bundled algorithm slighter different than in the literature [20] as no source code was available. Their Software City leaves all arcs visible but does not state how thick the lines are and how dense lines are packed in bundles. We used the same thickness as in DynaCity but had to use our taste when adjusting the line density and when mimicking the appearance to look like the ones in the literature. Line thickness and bundle density are no threats to validity as subjects were able to zoom in/out if they needed a closer look (or a bird's eye perspective).

Subjects. All subjects were employees of the same IT company and were working at the same site. As UML diagrams, representations of dependencies as arcs, hierarchical structures, etc., are universal concepts known by any professional software engineer anywhere, we are convinced that cultural differences do not pose a threat to validity.

Supervisor. Since the supervisor works in the same company as the subjects, they knew each other. This may potentially pose a risk, as the subjects could have wanted to help the supervisor support the theses and therefore intentionally performed worse when they were in the edge-bundled group. As a countermeasure, subjects did not know the origins of the two variants and did not know the hypotheses. Both groups were also encouraged to perform as well as possible. Note, that the edge-bundled group performed better or equal on some tasks (time spent on task #4, correctness of tasks #2 to #5). This also indicates that this threat is low (at most).

6. DynaCity$_{rc}$: Extension with color-encoded error/status codes

To determine the overall health of a system or performing a root cause analysis, it is crucial to visualize errors that occurred at runtime. This information is not available in the pure DynaCity visualization because the two visualizations serve completely different use cases. The pure DynaCity supports software engineers in program comprehension, i.e., understanding the functionality of the system, the internal structure, the dependencies between artefacts also at runtime, and so on. Coloring based on the HTTP status codes would shift the focus away from program comprehension towards error analysis, since software engineers would want to understand the error first, i.e., why a component is red. To avoid this distraction, we separate the two use cases, program comprehension and root cause analysis, into two separate visualizations: DynaCity and DynaCity$_{rc}$.

Even if fixing a bug has to happen quickly in everyday work and software engineers do not always have time to understand it in depth right away, it is crucial to find and understand the root cause at the latest in a follow-up. An efficient solution for a bug improves the quality of the software as a whole [37]. Since especially in distributed microservice architectures the root cause is difficult to find [38,39] the DynaCity$_{rc}$ visualization is to support software engineers during this process.

DynaCity$_{rc}$ enhances the visualization of traces by not only showing their intensities, but also displaying HTTP response error/status codes using a coloring scheme. Thus, the system's success/failure rate can be identified at a glance. When we initially showed the idea to others and asked if it is better to color just the arcs or the buildings and the arcs based on the HTTP status codes, we got ambiguous responses. While some said that coloring the buildings makes it much easier to spot the failures, others said that we might be implying a false root cause, as users no longer look at the arcs, but only at the more concise building colors. Therefore, we implemented two variants to determine the right amount of coloring: we either only color the arcs depending on the trace status code (see an example in Fig. 11(a)) or we color both the arcs and the buildings, see Fig. 11(b). Note that pure DynaCity is similar to variant 1 if there are no errors. Otherwise both variants are more helpful for root cause analyses than pure DynaCity, see Fig. 2. In its first variant DynaCity$_{rc}$ leaves the colors of the buildings in magenta, since this color is neutral and does not evoke positive/negative associations among users (as green or red would). The second variant also colors buildings w.r.t. the error codes. The idea is that with this users can quickly see which components are causing problems. The downside is that the visualization may be too colorful and thus may overwhelm the users. In our second controlled experiment, we examine which variant better assists professional software engineers when using DynaCity$_{rc}$ to perform root cause analyses.

6.1. Coloring

To determine not only the intensity but also the color, we need to classify the status code of a HTTP call according to the typical scheme (`2xx`: successful operation, `4xx`: client error, `5xx`: server error) and group calls accordingly. We omit the rare other groups. For each group we calculate the component load and the call load and afterwards apply an IQR normalization on the total load. As before, we map the resulting normalized load linearly to the brightness values. DynaCity$_{rc}$ uses the hue color wheel to derive an even color gradient. The hue of a color is measured as an angle around the circle with red being at 0°, yellow at 60°, and green at 120°. We map `2xx` status codes to 120°, client errors to 60°, and server errors to 0°. The reason for mapping client errors to a yellow color is that they may or may not be semantic errors, as for example a `401 Unauthorized` can be completely legitimate. With this mapping we then calculate the resulting color of both the call load (both variants) and the component load (variant 2 only) as:

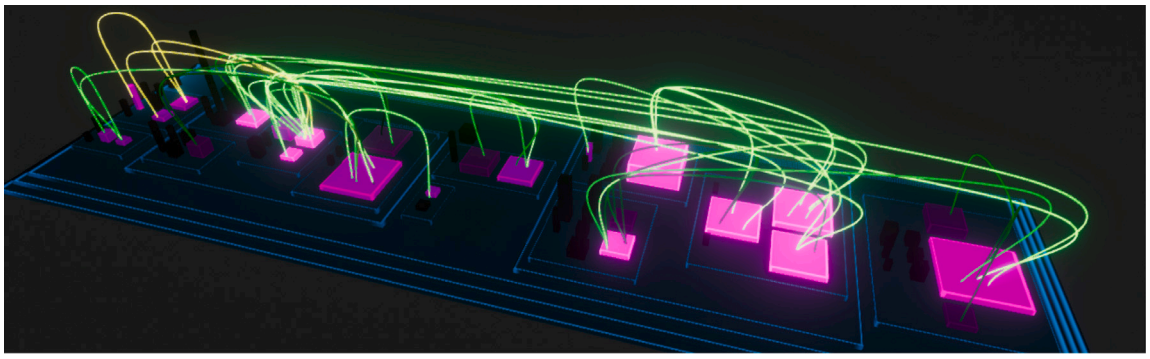
$$hueColor = \frac{load_{success} * 120 + load_{clientError} * 60 + load_{serverError} * 0}{load_{success} + load_{clientError} + load_{serverError}} \quad (4)$$

While the coloring is straightforward for the call load, it needs clarification for the component load. A component load is considered to be erroneous if it produces the error itself or if any of its spawned child calls does, because the HTTP status code always refers to the entire trace and not to individual spans, i.e. individual calls. We decided against mapping the error only to the last call, because often a wrong input of a parent call can be the reason for the error. By marking all calls of a faulty trace red, the engineer can follow the error propagation through the application. Otherwise the visualization may suggest a wrong cause by suppressing the other calls.

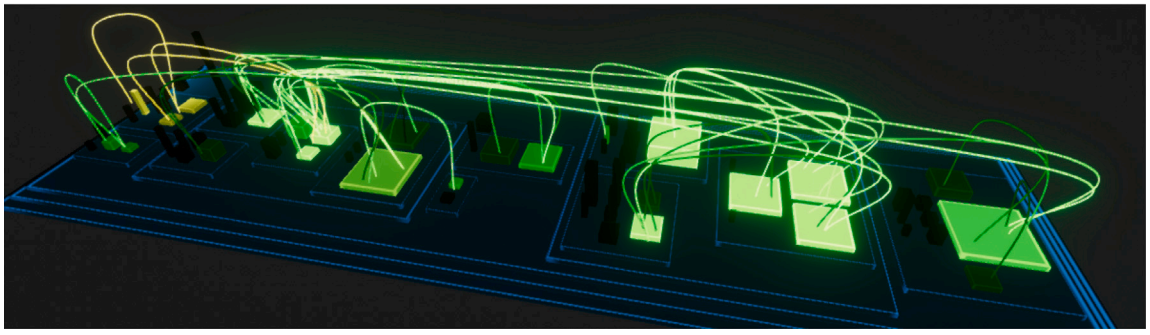
6.2. Controlled experiment on the DynaCity$_{rc}$ extension

To evaluate the usefulness of the two variants, we conducted a second study based on the blogging project known from Section 4.1.

Subjects. From the professional software developers who participated in the first study (see Section 4.3) a subset of 28 subjects were available for the second study. They had a similar average work experience (2–5 years) and were familiar with the visualization from the first study. We randomly assigned them to one of the two groups: 14 subjects performed the experiment with variant 1 (only arcs colored), hereafter called AC group, and 14 with variant 2 (arcs and buildings colored), hereafter called ABC group. We had again two female subjects in each group and the average work experience was equal between groups.



(a) DynaCity$_{rc}$ variant 1: Yellow arcs with error status (on the left); intensities of arcs and buildings as in pure DynaCity.



(b) DynaCity$_{rc}$ variant 2: Yellow arcs *and yellow buildings* (on the left); green buildings = no errors on incoming/outgoing arcs; intensities as in pure DynaCity.

Fig. 11. Color-encoded error/status codes in DynaCity$_{rc}$, two variants. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Research Questions. To identify the most common real-world root causes of failures, we interviewed five leading software architects of the same IT company that provided the subjects. They were different from the study subjects and knew nothing about the study. They all have the technical responsibility for Java-based software applications that have a microservice architecture and are used productively at customer sites. The most frequently mentioned root causes for anomalies in such software were:

#1. Database unreachable
#2. Incorrect configuration of the authorization/authentication
#3. Timeouts due to excessive load/amount of data

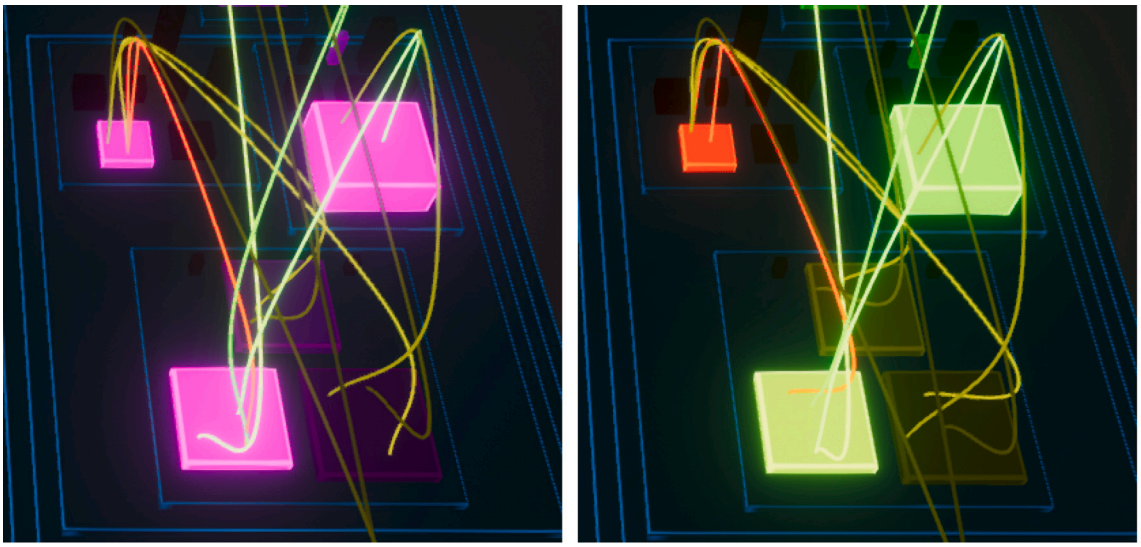
We had actually also expected network failures as a frequent cause of errors, but these were not mentioned by the experts. However, in terms of HTTP status codes, this error scenario would appear quite similar to error scenario #3, as network failures would also result in 503 or 504 HTTP errors, yielding similar trace data and hence a similar visualization.

In our second study, we investigate how well the two variants of error coloring help in root cause analyses in software that misbehaves and shows those three types of runtime errors. We examine the following two hypotheses:

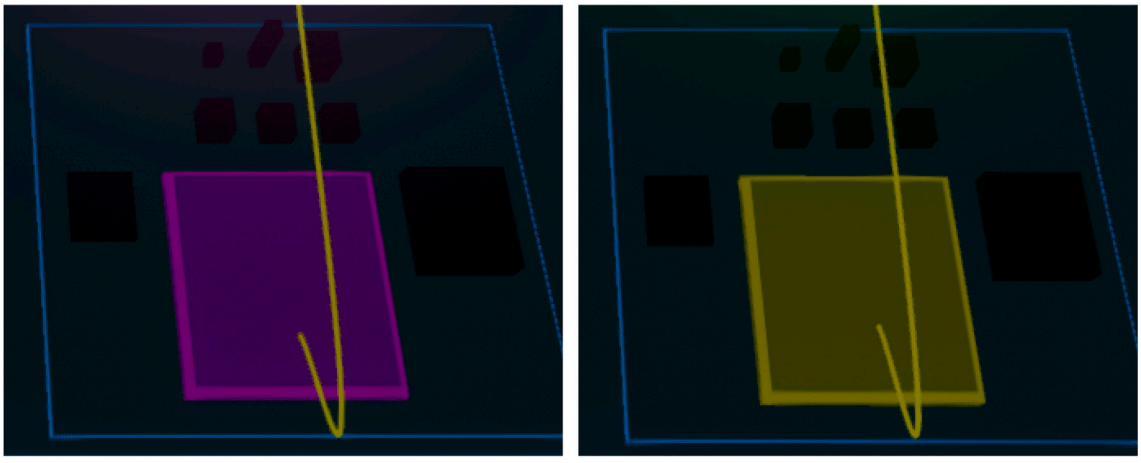
H4: The use of DynaCity$_{rc}$ is suitable for performing root cause analysis.
H5: The coloring of arcs and buildings is more helpful than only colored arcs.

Experiment. To answer our research questions, we simulated the three error scenarios in our study object. As a baseline we reused from Section 4.1 the 15 min of collected trace data where the application ran without any errors. Then for each of the three types of anomalies we artificially created the circumstances for the error scenarios. For scenario #1 we shut down the database by hand (resulting in 500 HTTP status codes). For scenario #2 we created a bug in the source code of the JWT token validation so that requests were rejected although they have a valid token (resulting in 403 HTTP status codes). To simulate scenario #3 we increased the application load, added a sleep function in the source code, and reduced the number of allowed concurrent connections in the application service (resulting in 504 HTTP status codes). We recorded the trace data of each faulty run (for 15 min). For the baseline data and for each of the three faulty scenarios we then created DynaCity$_{rc}$ visualizations, in two variants each. Participants in the AC group received the baseline visualization and the three error scenario visualizations from the left column of Fig. 12 (only arcs colored). The ABC group got the baseline plus the right column (arcs and buildings colored). The subjects were then instructed to perform a root cause analysis (RCA) on each scenario and spot the component(s) they believed to be responsible for the outage as well as assess the possible root cause. We measured the correctness of the answers and the time spent for each RCA. In a preliminary study, we found a time limit of three minutes per scenario to be reasonable. The study was again performed remotely. In order to answer our research questions the subjects again received an exit questionnaire after they completed all three RCAs. For testing our hypothesis H4 we asked how much the visualization helped them in performing the RCAs. For testing our hypothesis H5 we also showed screenshots of the respective other visualization variant and asked subjects to compare, if the other variant would have helped them more in performing the RCA. To test H5 in more detail, the subjects had to evaluate the other visualization variant with respect to (a) clarity, (b) comprehension, (c) support in solving the tasks, and (d) personal preference.

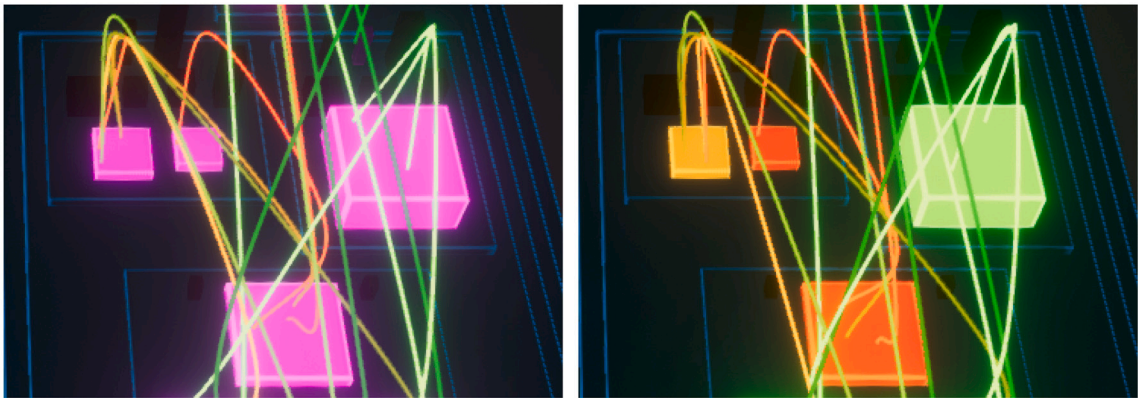
In our first experiment we already showed with the NASA TLX questionnaire that the cognitive load is low with the DynaCity visualization. Since in the second experiment both groups work with the DynaCity



(a) Scenario #1: Database unreachable.



(b) Scenario #2: Authentication misconfiguration.



(c) Scenario #3: Timeouts.

Fig. 12. Simulated error scenarios; variant 1 (only arcs colored) on the left, variant 2 (both arcs and buildings colored) on the right. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

visualization and only different colorings are studied, we took a low cognitive load for granted and did not re-examine this questionnaire.

Results and Discussion. From here on, statistical significance is always determined by a t-test with significance level $\alpha = 0.05\%$. Fig. 13 shows how many subjects correctly identified the component responsible for the faulty behavior and assessed the right root cause for each of the three anomalies. In the AC group 13, 9, and 6 subjects were right in finding the responsible component resulting in an average correctness of 66.7%. The ABC group had 14, 12, and 13 correctly identified faulty components for the three error scenarios, i.e., a better average correctness of 92.9%. The difference is also statistically significant. For the root cause detection the results are also statistically significant: For

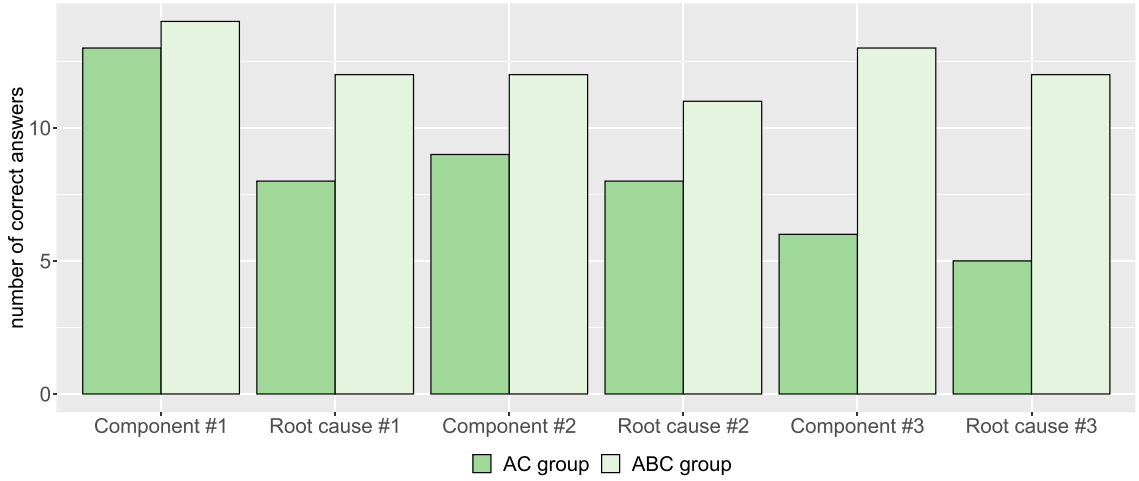


Fig. 13. Correctness of the RCA; AC group in dark green (left bars); ABC group in light green (right bars). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

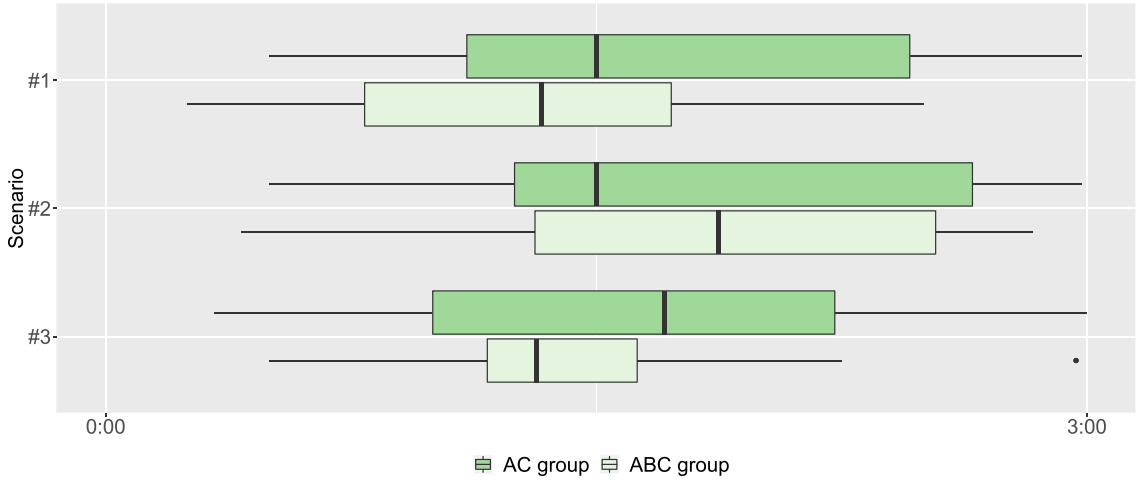


Fig. 14. Time spent on performing the RCA (in minutes). AC group in dark green on top and ABC group in light green below. Boxplots as in Fig. 5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the three error scenarios $8 + 8 + \mathbf{5}$ members of the AC group found the correct root cause (50.0% average), whereas the ABC group had $12 + 11 + \mathbf{12} = 83.3\%$ correct answers. The differences between the groups are particularly striking in error scenario #3 (compare the two rightmost bars in Fig. 13). For an error of this type that affects multiple components the coloring of the buildings seems to be especially helpful for the ABC group. In Fig. 12(c, right) the responsible components stand out prominently in orange and red and immediately point to the database components that are responsible for the timeouts. Since the alarming colors are missing in the other variant (Fig. 12(c, left)), the AC group struggled when pinpointing the root cause to the database components.

Fig. 14 shows the time measurements. The results are not statistically significant. The time difference in scenario #1 is marginal. The AC group solved scenario #2 faster. Here the subjects of the AC group instantly spotted the one yellow arc standing out in front of the dark buildings, see Fig. 12(b, left). It took the ABC subjects longer since they also analyzed other buildings that changed their colors between the baseline and the error visualization. On the other hand due to their more careful analysis they came to the correct root cause more often. The ABC group solved scenario #3 faster and as we discussed above also better, which we can again explain by the prominent orange and red database components, which also put the subjects onto the right track more quickly.

We also asked in the exit questionnaire of our second study:

1. "How much did the visualization help you in solving the tasks?", scale: 0 = little, 4 = much.
2. "Would the other visualization have helped you more in solving the tasks?", scale: 0 = much less, 4 = much more.
3. "Rate the *other* visualization variant in comparison with your assigned one with respect to (a) clarity, (b) comprehensibility, (c) support in solving the tasks, and (d) personal preference".

Fig. 15 shows the boxplots for the first two questions. Both groups stated that their assigned visualization helped them *much* in solving the tasks (first group of boxplots of Fig. 15). Hence, professional software engineers benefit from DynaCity$_{rc}$ when performing root cause analyses. Together with the correctness results (the ABC group identified the correct component in 92.9% and the correct root cause in 83.3%), we can accept H4, i.e., DynaCity$_{rc}$ supports performing RCAs.

Let us now test hypothesis H5 with questions 2 and 3. The answers to question 2 are given in the second group of boxplots of Fig. 15. The statistically significant result already is that buildings should also be colored because (a) the median of the AC group is convinced that the ABC variant would have helped them *more* (score = 3 of 5) in performing the RCA, and (b) the median of the ABC group assess the helpfulness of the AC variant *less* (score = 1 of 5). Hence, the helpfulness is 2/5 = 40% better.

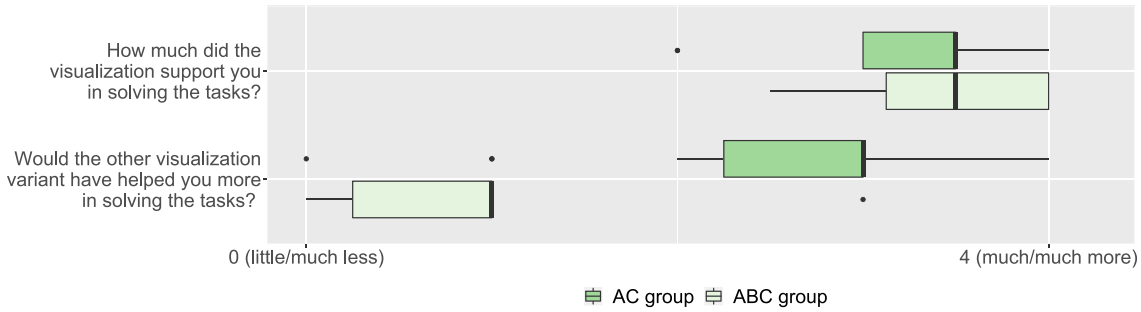


Fig. 15. Helpfulness of variants. Boxplots as in Fig. 5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

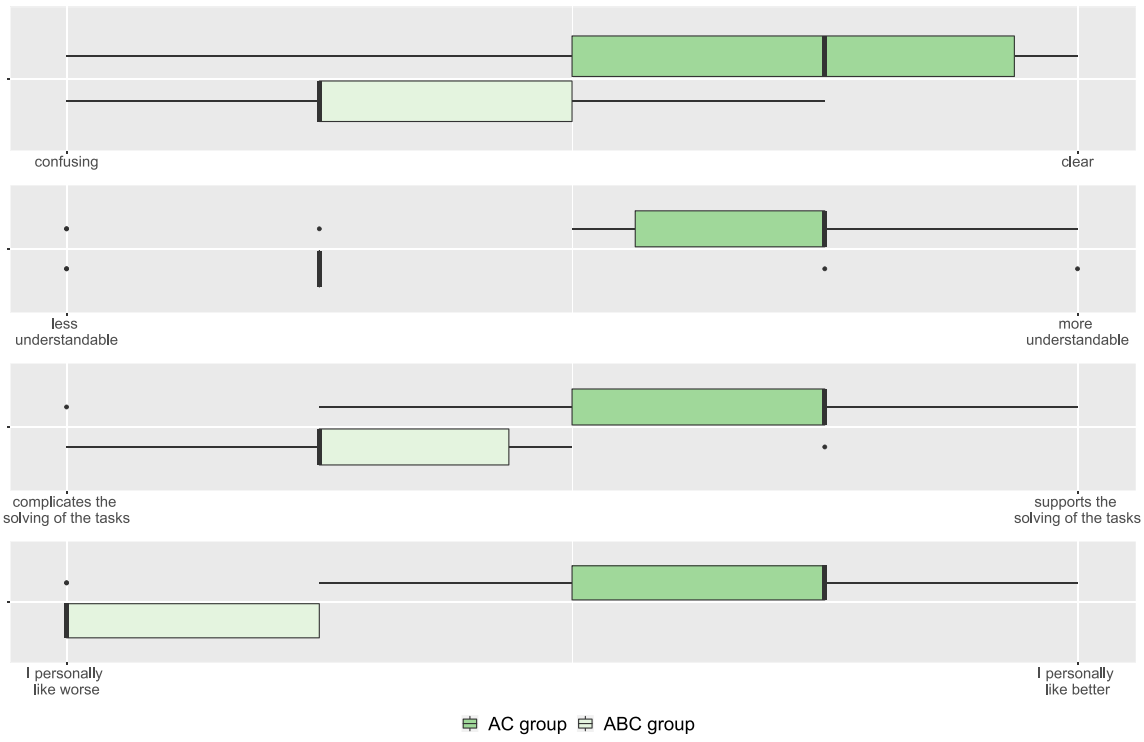


Fig. 16. Comparison of the assigned visualization variant and the other one in terms of fours metrics. The AC group rates the ABC variant and the ABC group rates the AC variant. Boxplots as in Fig. 5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Question 3 compares the variants in some more detail. Fig. 16 holds the results. The ABC group compared their assigned visualization (arcs and buildings colored) with the AC variant of the visualization (only arcs colored). This group did the root cause analyses with the ABC variant of the visualization and in the questionnaire they were shown pictures of the other variant and were asked if they think that their performance would have been better if they had the other variant. (Similar for the AC group that saw the ABC visualization at the time of the questionnaire.) All four boxplots of the ABC group are on the left side of the diagram, i.e., the ABC subjects saw the AC variant as rather confusing, less understandable, complicating the solving of tasks, and they liked it worse. In contrast, the boxplots of the AC group are all on the right: The AC subjects found the colored buildings rather clear, more understandable, supporting the solving of tasks, and they also liked it better. These differences are statistically significant. Together with (a) the correctness results (the ABC group was statistically significant 26.2% (= 11/42) better at finding the correct component and 33.3% (= 14/42) better at finding the root cause than the AC group) and (b) the clear preferences expressed in the answers to question #2 of the exit questionnaire, we can also accept H5, i.e., to also color the buildings is more helpful.

Threats to Validity. We assess the threats to validity of our second study as low. We already discussed the threats regarding subjects, object and supervisor in Section 5.5. We chose the error scenarios based on the expert interviews to ensure we are testing on real-world problems. Nevertheless, all experts are employed by the same company, so that we cannot completely exclude a bias. But Gunawi et al. [40] analyze hundreds of service outages and their root causes and also identify all the root causes we chose. Another potential threat is the magenta color of the buildings in the AC variant. It may be that when asking subjects which variant they like better (bottom-most boxplot of Fig. 16) their decision is *mainly* based on whether they prefer magenta buildings (variant 1) or (predominantly) green ones (variant 2). Although the personal color preference may affect this last boxplot, the other unaffected ones still are in favor of the additional coloring of buildings. Hence, we consider this risk to be low.

7. Conclusion

Huge amounts of runtime data (like traces) characterize the dynamic behavior of a software system. Visualizations help software engineers in analyzing the data, provided its amount is reduced first.

To reduce trace data, this paper presented a combination of both a summarization and a metrics-based approach. DynaCity aggregates all calls between the same two components into one representative arc in a Software City view and calculates its aggregated load based on the call frequencies and the call durations. An arc's brightness reflects the aggregated load. Compared to the state of the art, this reduces the number of arcs and makes DynaCity's visualization less cluttered. DynaCity also displays the activity of the component's buildings: the more and longer requests they receive and/or spawn, the brighter they are. The extension DynaCity_{rc}, specially targeted towards root cause analyses, colors the arcs and buildings according to the HTTP status codes of their traces, so that for instance unsuccessful requests lead to a red color.

In the first controlled experiment we posed typical software comprehension tasks to 30 professional software engineers. When using the DynaCity visualization (no error-encoding colors) compared to a traditional edge-bundled trace visualization they performed 11.7% better and saved 5.83% of the total allotted time while perceiving a 6.70% (percent points) lower/better overall cognitive workload. We could also show that software engineers prefer the DynaCity visualization. In the second controlled experiment we found with a subset of 28 subjects that DynaCity_{rc} helps much when performing root cause analyses. We could also show that developers found it 40% more helpful when not just arcs are colored based on HTTP status codes but also the buildings. Coloring both improves the correctness of finding the responsible component by 26.2% and of finding the root causes by 33.3%. The source code and the raw data of the quantitative evaluations are available from https://github.com/qaware/dynacity.

CRediT authorship contribution statement

Veronika Dashuber: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft, Review & editing. **Michael Philippsen:** Conceptualization, Methodology, Writing – original draft, Review editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Sridharan, Distributed Systems Observability, O'Reilly Media, Inc., Sebastopol, CA, 2018.

[2] M. Song, H. Yang, S. Siadat, M. Pechenizkiy, A comparative study of dimensionality reduction techniques to enhance trace clustering performances, Expert Syst. Appl. 40 (9) (2013) 3722–3737.

[3] F. Fittkau, J. Waller, C. Wulf, W. Hasselbring, Live trace visualization for comprehending large software landscapes: The ExplorViz approach, in: 1st IEEE Work. Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013, pp. 1–4.

[4] B. Cornelissen, A. Zaidman, A. van Deursen, A controlled experiment for program comprehension through trace visualization, IEEE Trans. Softw. Eng. 37 (3) (2011) 341–355.

[5] D. Holten, B. Cornelissen, J.J. van Wijk, Trace visualization using hierarchical edge bundles and massive sequence views, in: 4th IEEE Intl. Workshop Vis. Softw. for Understanding and Anal., Banff, Canada, 2007, pp. 47–54.

[6] J. Trümper, J. Bohnet, J. Döllner, Understanding complex multithreaded software systems by using trace visualization, in: Proc. 5th Intl. Symp. on Softw. Vis., Salt Lake City, UT, 2010, pp. 133–142.

[7] R. Wettel, M. Lanza, Visualizing software systems as cities, in: Proc. 4th IEEE Intl. Workshop on Vis. Softw. for Understanding and Anal., Banff, Canada, 2007, pp. 92–99.

[8] V. Dashuber, M. Philippsen, Trace visualization within the software city metaphor: A controlled experiment on program comprehension, in: Proc. IEEE Work. Conf. on Softw. Vis., Online, 2021, pp. 55–64.

[9] J. Waller, C. Wulf, F. Fittkau, P. Dohring, W. Hasselbring, Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency, in: Proc. 1st IEEE Work. Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013, pp. 1–4.

[10] B. Cornelissen, L. Moonen, A. Zaidman, An assessment methodology for trace reduction techniques, in: Proc. IEEE Intl. Conf. on Softw. Maintenance, Beijing, China, 2008, pp. 107–116.

[11] W.D. Pauw, D. Lorenz, J. Vlissides, M. Wegman, Execution patterns in object-oriented visualization, in: Proc. 4th USENIX Conf. on Object-Oriented Technologies and Systems, Santa Fe, NM, 1998, pp. 219–234.

[12] B. Cornelissen, A. van Deursen, L. Moonen, A. Zaidman, Visualizing testsuites to aid in software understanding, in: Proc. 11th Europ. Conf. on Softw. Maintenance and Reengineering, Amsterdam, The Netherlands, 2007, pp. 213–222.

[13] A. Hamou-Lhadj, T. Lethbridge, Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system, in: Proc. 14th IEEE Intl. Conf. on Program Comprehension, Athens, Greece, 2006, pp. 181–190.

[14] A. Rohatgi, A. Hamou-Lhadj, J. Rilling, An approach for mapping features to code based on static and dynamic analysis, in: Proc. 16th IEEE Intl. Conf. on Program Comprehension, Amsterdam, The Netherlands, 2008, pp. 236–241.

[15] A. Zaidman, S. Demeyer, Automatic identification of key classes in a software system using webmining techniques, J. Softw. Mainten. Evol.: Res. Pract. 20 (6) (2008) 387–417.

[16] A. Knupfer, H. Brunst, W. Nagel, High performance event trace visualization, in: Proc. 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, 2005, pp. 258–263.

[17] S. Maoz, A. Kleinbort, D. Harel, Towards trace visualization and exploration for reactive systems, in: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing, Coeur d'Alene, ID, 2007, pp. 153–156.

[18] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, A. van Deursen, Understanding execution traces using massive sequence and circular bundle views, in: Proc. 15th IEEE Intl. Conf. on Program Comprehension, Banff, Canada, 2007, pp. 49–58.

[19] S.v.d. Elzen, D. Holten, J. Blaas, J.J. van Wijk, Dynamic network visualization with extended massive sequence views, IEEE Trans. Visual. Comput. Graphics 20 (8) (2014) 1087–1099.

[20] P. Caserta, O. Zendra, D. Bodenes, 3D hierarchical edge bundles to visualize relations in a software city metaphor, in: Proc. 6th Intl. Workshop on Vis. Softw. for Understanding and Anal., Williamsburg, VA, 2011, pp. 1–8.

[21] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, A. Matsuo, SArF map: Visualizing software architecture from feature and layer viewpoints, in: Proc. 21st IEEE Intl. Conf. on Program Comprehension, San Francisco, CA, 2013, pp. 43–52.

[22] O. Benomar, H. Sahraoui, P. Poulin, Visualizing software dynamicities with heat maps, in: Proc. 1st IEEE Work. Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013, pp. 1–10.

[23] A. Krause, M. Hansen, W. Hasselbring, Live visualization of dynamic software cities with heat map overlays, in: Proc. IEEE Work. Conf. on Softw. Vis., Online, 2021, pp. 125–129.

[24] P. Dugerdil, S. Alam, Execution trace visualization in a 3D space, in: Proc. 5th Intl. Conf. on Information Technology, Las Vegas, NV, 2008, pp. 38–43.

[25] J. Vincur, P. Navrat, I. Polasek, VR city: Software analysis in virtual reality environment, in: Proc. IEEE Intl. Conf. on Softw. Quality, Reliability and Security Companion, Prague, Czech Republic, 2017, pp. 509–516.

[26] S. Alam, P. Dugerdil, EvoSpaces visualization tool: Exploring software architecture in 3D, in: Proc. 14th Working Conf. on Reverse Eng., Vancouver, Canada, 2007, pp. 269–270.

[27] P. Caserta, O. Zendra, Visualization of the static aspects of software: A survey, IEEE Trans. Vis. Comput. Graph. 17 (7) (2011) 913–933.

[28] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, L. Su, Graph-based trace analysis for microservice architecture understanding and problem diagnosis, in: Proc. 28th ACM Joint Mtg. on Europ Softw. Eng. Conf. and Symp. on the Foundations of Softw. Eng., ACM, Virtual Event USA, 2020, pp. 1387–1397.

[29] M. Szvetits, U. Zdun, Enhancing root cause analysis with runtime models and interactive visualizations, in: Proc. 8th Intl. Workshop on MoDELS@Run.Time, Miami, FL, 2013.

[30] S. Tricaud, Picviz: Finding a needle in a haystack, in: Proc. 1st USENIX Conf. on Anal. of Sys. Logs, San Diego, CA, 2008, pp. 3–12.

[31] X. Li, M.C. Huang, K. Shen, L. Chu, A realistic evaluation of memory hardware errors and software system susceptibility, in: Proc. USENIX Annual Technical Conf., Boston, MA, 2010, pp. 75–88.

[32] U. Sedlar, M. Volk, J. Sterle, A. Kos, R. Sernec, Contextualized monitoring and root cause discovery in IPTV systems using data visualization, IEEE Network 26 (6) (2012) 40–46.

[33] T. Takada, H. Koike, Tudumi: Information visualization system for monitoring and auditing computer logs, in: Proc. 6th Intl. Conf. on Information Vis., London, United Kingdom, 2002, pp. 570–576.

[34] V. Dashuber, M. Philippsen, J. Weigend, A layered software city for dependency visualization, in: Proc. 16th Intl. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications, Online, 2021, pp. 15–26.

[35] M. Pacione, M. Roper, M. Wood, A novel software visualisation model to support software comprehension, in: Proc. 11th Work. Conf. on Reverse Engineering, Delft, The Netherlands, 2004, pp. 70–79.

[36] S.G. Hart, L.E. Staveland, Development of NASA-TLX (task load index): Results of empirical and theoretical research, in: Human Mental Workload, Elsevier, Amsterdam, The Netherlands, 1988, pp. 139–183.

[37] H. Lal, G. Pahwa, Root cause analysis of software bugs using machine learning techniques, in: Proc. 7th Intl. Conf. on Cloud Computing, Data Science & Eng., Noida, India, 2017, pp. 105–111.

[38] M. Jin, A. Lv, Y. Zhu, Z. Wen, Y. Zhong, Z. Zhao, J. Wu, H. Li, H. He, F. Chen, An anomaly detection algorithm for microservice architecture based on robust principal component analysis, IEEE Access 8 (2020) 226397–226408.

[39] L. Wu, J. Tordsson, E. Elmroth, O. Kao, MicroRCA: Root cause localization of performance issues in microservices, in: Proc. Network Operations and Management Symposium, Budapest, Hungary, 2020, pp. 1–9.

[40] H.S. Gunawi, M. Hao, R.O. Suminto, A. Laksono, A.D. Satria, J. Adityatama, K.J. Eliazar, Why does the cloud stop computing?: Lessons from hundreds of service outages, in: Proc. 7th ACM Sympo. on Cloud Computing, Santa Clara CA USA, 2016, pp. 1–16.