Software Development, Maintenance and Operations: Course projects

Alexander Bakhtin, Matteo Esposito

Autumn 2025

General information

Each student is required to participate in a project to pass the course. Course grading is based mostly on project performance.

Projects are implemented in groups of **2-3 people** for a maximum of **6 people**. Exceptionally, implementing the project individually is also possible.

Students are required to register their group in Moodle to enable submission and grading on behalf of the entire group, even if working alone.

General requirements and submissions for all projects:

- Implement the code in a clean and modular way, with appropriate functions that perform a single, well-defined task. The submission should contain the entire codebase and data files (i.e., the git repository of the project) as a subdirectory of the submission archive. The repository should contain enough documentation so that teachers can execute all the stages of the analysis and confirm the implementation and results.

- Implement some unit tests for several functions in your code. The submission should contain a separate document with a test plan, stating which parts of the project were tested and what the test coverage is. The testing code should be part of the submitted repository, and course teachers should be able to execute it. In the project report, discuss whether this process helped you catch bugs or somehow improved the structure and development process of the project.

- Test the quality of your project code with a tool **like SonarQube**. Attach a quality report from such a tool to your submission. If possible, address certain issues highlighted by the quality analysis and submit before-and-after reports from the tool. In the project report, discuss whether this process helped you discover bad practices in your code and improve it.

- Project report. The project report should describe the entire development and data analysis processes. Describe where you got the data from, its properties, how it was analyzed, and what observations and conclusions

you can make based on the analysis. Appropriate citations should be put for datasets and research papers. Each project contains its own list of requirements to mention in the report. Also reflect on the previous points concerning testing and quality analysis.

All report documents must be prepared in the IEEE conference template, either with LaTeX(Overleaf) or Microsoft Word [1].

After the project is submitted, in the **peer-review** (peer feedback) phase, each team member will be asked to evaluate the contributions of other team members across several categories. Students' own averaged evalutions will contribute up to 10% of the final grading points.

Project 1: Identifying unique developers in OSS repositories

Mining commit data provides us with the name and email of the author/committer as they were set in that user's git configuration. This creates an issue of duplicate developer identities in the data, since the same person might forget to properly configure this data on different machines or simply change emails (or names). Thus, given the full list of developer names and emails from mined commit data, we are required to perform developer de-duplication.

Many existing works constructing developer social networks through mining software repositories do not deal with this issue and do not even address it as a threat to the validity of the work [2], apparently treating each unique record as a unique developer. Some authors leveraged proprietary industrial data, where arguably more strict rules regarding configurations and committing to projects were in place and identity of each developer can be reliably established [8], while others gathered data from official database dumps, which include a *users* table acting as a foreign key for other records involving users [12]. Finally, some works only gathered a small set of developers, so identification of inconsistencies and merging of identities were performed manually [3].

Established heuristics of identifying unique personalities from such mined data do exist; however, they date back 20 years and concern the mining of mailing lists. The problems of identifying unique individuals from commit information and emails are related due to the fact that, in both cases, we only obtain a *(name, email)* pair. The heuristic by Bird et al. [7] is the most commonly cited one.

The following discrepancies in the data can occur, leading to duplication of developer information [9]:

- Reordering of parts of name
- Misspellings
- Non-ASCII characters, such as other alphabets, diacritics, and others

[1] `https://template-selector.ieee.org/`

- Use of several parts of a name, such as middle name, patronymic or additional surnames, their shortenings to middle initials, as well as different punctuation
- Using a username or diminutive names instead of the full first name
- Mistakes in Git configuration leading to artifacts, such as a commit message appended to the name

Bird et al. [7] heuristic The pairs of *(name, email)* are pre-processed to remove accents, convert to lowercase, remove duplicates, trailing, and leading whitespaces. The *name* is considered to consist of space-separated *firstname* and *lastname*, while the email is considered to take the form *prefix@domain*. Denoting the Levenshtein similarity function as *sim()* and the desired similarity threshold as t, we check two items $(name_1, email_1)$ and $(name_2, email_2)$ for the following conditions:

C1: $sim(name_1, name_2) \geq t$;

C2: $sim(prefix_1, prefix_2) \geq t$;

C3: $sim(firstname_1, firstname_2) \geq t$ AND $sim(lastname_1, lastname_2) \geq t$

C4: $prefix_2$ contains the initial of $firstname_1$ and the entire $lastname_1$

C5: $prefix_2$ contains the initial of $lastname_1$ and the entire $firstname_1$

C6: $prefix_1$ contains the initial of $firstname_2$ and the entire $lastname_2$

C7: $prefix_1$ contains the initial of $lastname_2$ and the entire $firstname_2$

Two pairs are considered duplicates if any of the conditions are met.

Project steps

The course project repository provides an **example baseline implementation of the Bird heuristic**. The provided code takes a public GitHub project, mines its commits, extracts all *(name, email)* pairs, and applies the Bird heuristic to identify duplicate pairs.

For this project, implement the following:

1. Each team member selects a project on GitHub with at least 1000 commits and 1000 contributors.
2. Use the provided implementation to mine the projects' commits and apply Bird heuristic.
3. Select the threshold t so that the output contains a significant amount of pairs but is feasible to evaluate manually (500-1000 pairs).

4. Go through the output and evaluate if the identified pairs indeed indicate the same developer or not. Take note of how many are True Positives (TP) or False Positives (FP) and what causes the problems in identification.

5. Propose your own method to identify duplicate developers, aiming to achieve a better TP to FP ratio (i.e., better accuracy and recall). This method could be:

   - Adding/Removing conditions from the Bird heuristic
   - Leveraging different edit distance/similarity measures or different language processing methods altogether, such as Latent Semantic analysis [9].
   - Using Large Language Models through tools such as LM Studio [2]

6. Manually validate the results similarly to Bird heuristic and comment on the differences.

Reporting

In the report, include at least the following:

- Selected project(s) and their description and statistics (number of commits, releases, contributors, starts, etc)
- Statistics and results of using the Bird heuristic: number of *(name, email)* items and thus the total number of pairs that will be compared, selected threhold t, number of pairs given in the output, number of TP and FP after manual evaluation, summary of reasons why certain pairs were FP.
- Description of your own designed method.
- Results of using the designed method (same as for Bird heuristic). Did your method solve some of the issues you identified with the Bird heuristic?
- Compare the results between different analyzed projects. Were there differences between projects that affected the performance of either method?

Microservice tracing (Projects 2 & 3)

Microservice architecture (MSS) is a software development paradigm where a platform is split into independent sub-programs called *microservices* [11]. Such

[2] `https://lmstudio.ai/`, note that in this case you are dealing with other persons' personal data, so we advise against using online tools that can be training on submitted data.

an architecture gives rise to specific problems and anti-patterns, leading to architectural degradation [5]. The network of microservice communications, sometimes called the Service Dependency Graph (SDG), can detect some of these problems [5, 1]. Since microservices communicate in real-time, methods utilizing *temporal* networks, that is, considering networks that evolve with time, could be applied to address the problem of microservice architectural degradation.

In MSS research and development, we must execute complex and large systems to collect tracing data to analyze their behavior, gather insights, and test new and emerging ideas. Nonetheless, executing and collecting relevant non-trivial data is a lengthy process that requires considerable effort. Despite the daunting need for public and real-world-like datasets, researchers investigated a few related ones in the literature and executed similar systems every time.

A particular domain where microservice (MS) execution is needed is anomaly detection. A microservice anomaly generally refers to unexpected behaviors or performance issues within a microservices architecture that deviates from normal behavior [20, 16]. Anomalies can manifest as increased latency, resource exhaustion, unexpected failures, or performance degradation, often resulting from complex interactions within the distributed system [19, 18].

These anomalies can be generated through various reliability testing practices. For example, in fault injection testing, the system is introduced with faults such as network issues, hardware exhaustion, system crashes, and configuration errors [15]. Other practices include stress testing and endurance testing, which refer to running the system under a heavy load or long duration, respectively. Current research has placed emphasis on such performance-related anomalies.

Traces, logs, and metrics are fundamental for anomaly detection in MSS [18]. Anomaly detection requires the availability of large datasets containing data from different sources, including the log or traces of each service individually or metrics describing the performance of the infrastructure or an individual service container. Traces capture the flow of requests as they move through service instances (i.e., spans) within the MSS [3]. Logs record execution details of service instances. Metrics are quantitative measurements that track the performance of an entity, such as MS, or the entire host system over time.

Although MSS datasets that include all three modalities have been openly published [18, 10, 17], most of them either provide data on OSS benchmark ([18, 10, 13]) or proprietary, closed systems ([13, 17]. Other datasets do not provide all three modalities [19, 13].

Microservice tracing data is an important aspect of monitoring microservice systems. Tracing data allows the researcher/practitioner to see invocation chains of services and analyze how different services and their latency or stability affect other services and the status and health of the overall microservice system.

Tracing data comes naturally in the form of a temporal network, in particular, an edge flow network, i.e., the trace data represents that service A called service B at time T (with latency L). Such data can be analyzed leveraging Temporal Network methods, i.e., methods from a branch of Network Science

[3] `https://betterstack.com/community/guides/observability/what-is-opentelemetry/`

that investigates complex networks evolving with time.

Project 2: Time-dependent metrics and centrality for monitoring of microservice systems

Beres et al. [6] designed an algorithm that computes the centrality scores of nodes in a temporal network given as a stream of connections. The scores are updated whenever a new connection is registered. Centrality scores are used in network science to measure how important/influential a node is within a network [4]. The aim is to investigate if the real-time centrality score can function as a novel metric for microservice monitoring by correlating the value of centrality with latency or other metrics [4]. The metric could highlight problematic services that receive too many requests and thus need to be scaled-up or potentially refactored into several services.

Project steps

The course project repository provides a demonstration script that takes a sample of microservice traces [18], constructs an edge flow list, and computes the online temporal centrality values [6].

For this project, implement the following:

1. Identify a dataset of microservice monitoring data that includes both execution traces as well as metrics for each microservice[4].
2. Convert the traces to a format similar to the one used in the demonstration script, i.e. *(calling service, called service, timestamp)*.
3. Perform temporal centrality analysis to get the centrality time series for each service.
4. Test each metric and centrality time series for Normal distribution, exclude the ones that cannot be compared with temporal centrality (i.e., if most metrics are non-normal, exclude the normal ones).
5. Compute correlations between each metric and centrality time series.

Reporting

In the final report, include at least the following things:

- Description of the dataset with citation: what is the system used for data collection, how many microservices, what was the data collection period, and how much data was collected.

[4]Note that metrics and traces should cover the same time interval; in the data used in the example, this is not the case.

- If the dataset is too big, describe what subset of data you used for the project analysis.
- The results of the normal distribution test: what was the chosen significance level, how many metrics were normally distributed or otherwise, which metrics were excluded from further analysis, if any.
- Results of the correlation test: which correlation test was chosen (based on normality test), what was the significance level, how many statistically significant results, and which are the most positively or negatively correlated metrics. Create appropriate plots demonstrating temporal centrality and other metrics for some services.

Project 3: Anomaly detection in microservice systems using traces

Masuda and Holm [14] designed a method to cluster the different time intervals of a temporal network to identify intervals when a network appears to be in a certain state and when this state changes. The aim is to observe whether the states detected from a temporal networks of microservice traces can be identified with the anomalies in the system, given the ground truth of the induced anomalies. Leveraging this method could provide a lightweight, purely trace-based method for microservice anomaly detection.

Project steps

The course project repository provides a demonstration script that takes a list of traces, aggregates them into intervals of given length, and performs state detection [14].

For this project, implement the following:

1. Identify a dataset of microservice monitoring data that includes execution traces gathered while anomalies were injected into the system and the anomaly injection timeline.
2. Aggregate the traces to network snapshot using a window of a specific length and convert the data to a format similar to the demonstration script, i.e. *(calling service, called service, snapshot)*. Each team member should use a different length and perform their own analysis.
3. Perform system state detection to get the activity profile for each state. Experiment with the analysis, e.g. use different available distance functions (or implement your own) and identify different number of states. Each team member should perform their own analysis.
4. Attempt to match identified states with anomalies injected into the system, or evaluate if any state corresponds well to any anomaly. You can

find a method or tool to perform this (HINT: you could try leveraging this tool for this `https://tut-arg.github.io/sed_eval/index.html`. Even though it is designed for audio processing, it has the functionality to compare detected event intervals to the ground truth intervals, i.e., Sound Event Detection. Course staff have no experience with this tool, though, and support is not provided at the Q/As.)

Reporting

In the final report, include at least the following things:

- Description of the dataset with citation: what is the system used for data collection, how many microservices, what was the data collection period, how much data was collected, what were the injected anomalies.
- If the dataset is too big, describe what subset of data you used for the project analysis.
- The details of state detection: lengths of aggregation windows, parameters given to state detection (distance functions and their own parameters); if implemented own distance function or used methods other than Dunn Index (see demo script and [14]) to find the best clustering - define them.
- Results of the evaluation: could any of the detected states be identified with any of the injected anomalies with good precision for some of the aggregation windows.

References

[1] Abdullah Al Maruf et al. "Using microservice telemetry data for system dynamic analysis". In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2022, pp. 29–38.

[2] Dario Amoroso d'Aragona et al. "One microservice per developer: is this the trend in OSS?" In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2023, pp. 19–34.

[3] Alexander Bakhtin, Xiaozhou Li, and Davide Taibi. "Temporal Community Detection in Developer Collaboration Networks of Microservice Projects". In: *European Conference on Software Architecture*. Springer. 2024, pp. 174–182.

[4] Alexander Bakhtin et al. "Network centrality as a new perspective on microservice architecture". In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE. 2025, pp. 72–83.

[5] Alexander Bakhtin et al. "Survey on Tools and Techniques Detecting Microservice API Patterns". In: *SCC*. 2022.

[6] Ferenc Béres et al. “Temporal walk based centrality metric for graph streams”. In: *Applied network science* 3.1 (2018), p. 32.

[7] Christian Bird et al. “Mining email social networks”. In: *Proceedings of the 2006 international workshop on Mining software repositories*. 2006, pp. 137–143.

[8] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. “Mining and visualizing developer networks from version control systems”. In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 24–31. ISBN: 9781450305761.

[9] Erik Kouters et al. “Who’s who in Gnome: Using LSA to merge software repository identities”. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 592–595.

[10] Cheryl Lee et al. “Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data”. In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1750–1762.

[11] James Lewis and Martin Fowler. *Microservices: a definition of this new architectural term*. 2014. URL: `https://martinfowler.com/articles/microservices.html`.

[12] Ning Li, Wenkai Mo, and Beijun Shen. “Task recommendation with developer social network in software crowdsourcing”. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2016, pp. 9–16.

[13] Zeyan Li et al. “Actionable and interpretable fault localization for recurring failures in online service systems”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 996–1008.

[14] Naoki Masuda and Petter Holme. “Detecting sequences of system states in temporal networks”. In: *Scientific reports* 9.1 (2019), p. 795.

[15] Joshua Owotogbe et al. *Chaos Engineering: A Multi-Vocal Literature Review*. 2024. arXiv: `2412.01416 [cs.SE]`. URL: `https://arxiv.org/abs/2412.01416`.

[16] Murugiah Souppaya, John Morello, and Karen Scarfone. *Application Container Security Guide*. Tech. rep. SP 800-190. National Institute of Standards and Technology (NIST), 2017. URL: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf`.

[17] Tsinghua University. *AIOps Challenge 2021*. `https://www.aiops.cn/gitlab/aiops-nankai/data/trace/aiops2021`. 2021.

[18] Guangba Yu et al. “Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 553–565.

[19] Chenxi Zhang et al. “Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning”. In: *Proceedings of the 44th international conference on software engineering*. 2022, pp. 623–634.

[20] Yuan Zuo et al. “An Intelligent Anomaly Detection Scheme for Micro-Services Architectures With Temporal and Spatial Data Analysis”. In: *IEEE Transactions on Cognitive Communications and Networking* 6.2 (2020), pp. 548–561. DOI: 10.1109/TCCN.2020.2966615.