CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Microservice Granularity and Development Overhead

How Organisational Processes Can Impact Technical Debt

Master's thesis in Computer science and engineering

PHILIP WARFVINGE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Microservice Granularity and Development Overhead

PHILIP WARFVNGE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Microservice Granularity and Development Overhead
How Organisational Processes Can Impact Technical Debt
PHILIP WARFVINGE

Supervisor: Hamdy Michael Ayas, Computer Science and Engineering
Advisor: Michael Lawson, AstraZeneca
Examiner: Hans-Martin Heyn, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Microservice Granularity and Development Overhead
How Organisational Processes Can Impact Technical Debt
PHILIP WARFVINGE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Microservices have become more prevalent recently, with several large enterprises adopting them. Microservices promise many advantages, but there are also challenges involved with deciding on their granularity, and this decision has many effects. The thesis explored through an investigative case study whether practitioners at one organisation perceived that a finer-grained microservice architecture resulted in additional work, and it was concluded that they did. Notably, they pointed out that intentionally suboptimal granularity decisions were frequently taken in the interest of time for services with rapidly changing requirements, and that this in part was due to the additional difficulty of splitting the service. Quantitative data from these practitioners' organisation back this up. Other key results supported by both quantitative and qualitative data from the organisation are that the type of the microservice impacts how the granularity changes over time — that a small subset of microservices with rapidly changing requirements grow large, while most microservices tend to stay the same size after initial development work is done. It is then discussed how this growth can be seen as architectural technical debt caused partly by organisational processes, and that it would be beneficial for the organisation to consider this when planning. This adds further support to literature claiming that organisational processes are crucial for microservice development and that there are additional potential disadvantages beyond the immediately obvious to using microservices, and that it is crucial to consider carefully whether the advantages truly outweigh the disadvantages before choosing to use a microservice architecture, especially for smaller systems.

Keywords: software engineering, architecture, technical debt, microservice, granularity, overhead, extra work

Acknowledgements

Thank you Hamdy Michael Ayas for your invaluable feedback, constant encouragement and incredible patience, without which I would not have been able to finish this thesis. Also, Hans-Martin Heyn, you have my gratitude for your patience. Thank you, Michael Lawson and Amir Tohidi for your good support and pleasant discussions.

Philip Warfvinge, Gothenburg, August 2024

Contents

List of Figures

List of Tables

1
Introduction

In recent years, many large companies have started using microservice architectures (MSAs), systems consisting of small independently deployable units that can be maintained by a single team, and the benefits of them, such as improved scalability, are well documented [1, 2]. One thing that needs to be decided is how granular to make these services [3]. If they are made very large, then they are essentially smaller monoliths that likely cannot be deployed independently or managed by a single team nor be scaled independently from the rest of the application. If they on the other hand are made very small, then this is likely to lead to greater communication between these services. However, there is no universally agreed-upon definition of what "small" means, or how small is sufficiently small [3]. Several methods to decide on granularity have been proposed. One way is through domain-oriented design [4]. Several studies have investigated the performance impact of splitting microservices in a too fine-grained manner [5], which is an additional factor that could be considered when making such decisions. However, personal experience in the software engineering field has indicated that the development of many services comes with its own challenges, especially in configuring the communication for these services and interacting with other teams and stakeholders.

Furthermore, literature has covered how to migrate from monoliths to microservices, both at a technical and organisational level, and which decision points organisations face when doing so [6]. These migrations are often large projects and take a significant amount of time — but the software has to remain functional and evolve during and after this migration. The MSAs are designed to have specific quality attributes, but as any software grows it invariably deviates from them [7]. This thesis also investigates what causes this to happen in the context of microservice granularity.

Adopting microservices comes with extra costs, such as additional communication [4]. There is therefore a need to find a good balance of microservice granularity, which provides the benefits of using microservices without incurring too much additional work. However, how microservice granularity in particular affects the additional work is not well studied. Therefore, to be able to find an appropriate microservice granularity, it is necessary to understand the additional work associated with implementing a microservice architecture.

To improve the understanding of the relation between development overhead and granularity of microservices, this thesis investigates what practitioners perceive as development overhead, and how this overhead differs between different levels of microservice granularity within an IT landscape through an interpretive case study [8]. The study was conducted at a single large pharmaceutical company, where quantitative data is extracted from git repositories and qualitative data is gathered

through interviews with practitioners, such as about why they decided to make the architectural decisions they did—particularly about granularity.
The purpose of the study was to investigate how developers at the organisation perceive the additional work—the work that does not directly contribute to the core functionality of the software, i.e. development overhead. It will investigate what the developers consider to be overhead, what causes it, and what relation it may have to microservice granularity.
To achieve this purpose, three research questions are posed. The first research question relates to constructing a taxonomy for microservices with granularity and application area dimensions. The second is about practitioners' perceptions of development overhead and granularity. The third is about whether there is a difference in granularity, overhead, and the relation between them for different types of applications.

RQ1 *What categories of services exist within the organisation?* Since the purpose is to describe how development overhead relates to granularity, and there is no standard definition of granularity, as mentioned by Vera-Rivera et al. [9], it is necessary first to define what the thesis means by granularity. A way to do this is to create a taxonomy of services based on the services that exist within the IT landscape. It is unclear what the most useful metrics to categorize are, so this will also need to be investigated. Answering this question will make it easier to investigate the differences between various levels of granularity in microservices and their relation to other factors.
RQ2 *What is the development overhead as perceived by the practitioners within the organisation?* Since there are as stated potential non-trivial factors that affect development overhead, it is useful to know what practitioners consider development overhead, and how they perceive that it differs depending on the granularity of the microservice architecture. Qualitatively answering this question will enable further quantitative analysis as described in *RQ3* since additional metrics might become apparent.
RQ3 *What is the relation between development overhead, microservice granularity and application area within the organisation?* To achieve the thesis' purpose, it would be of value to combine the results of RQ2 and quantitative analysis and compare how the perception relates to the actual data.

The thesis investigates these questions by performing a thematic analysis based on data gathered from eight interviews at the organisation. The result of this analysis is a taxonomy of service types and development overhead. Then, to further answer RQ3, quantitative data from git repositories is statistically analyzed to check whether the practitioner's claims are also supported by the data.

2

Theory

2.1 Microservices

The term "microservices" gained popularity after it was introduced by Fowler and Lewis in 2014 [2], and then gained significant traction after a successful transition to microservices at Netflix was presented by Cockroft in the GOTO 2014 conference [1]. After this it has been adopted by many leading technology companies such as Amazon [10] and Uber [11]. Fowler and Lewis presented some key characteristics of microservices, such as componentization via services, organizing services around business capabilities and decentralized governance, but they did not provide a concrete definition.

In general, microservices are small, independently deployable units of code, that decompose a business domain into a bounded context that a single team can manage. Commonly they are interacted with by users and do inter-service communication through REST APIs or asynchronous messaging.

Microservices are prime candidates for containerization, a form of lightweight virtualisation, commonly utilized through Docker. This, in combination with proper design, such as ensuring low coupling by avoiding excessive inter-service communication [12, 13], provides benefits such as increased availability, better fault tolerance, better scalability and improved elasticity due to greater ease of horizontal scaling, i.e. the ability to scale up and down by adding or removing additional computational units, rather than increasing the resources available to existing units.

2.2 Technical Debt

The term Technical Debt was introduced by Ward Cunningham [14] and has since been thoroughly used within the industry and academia [15]. He relates the concept of not doing things "the right way" in the interest of time to the concept of a financial loan. He describes that debt is accumulated over time and notes the importance of compound interest—that the rate at which it grows will increase, and that it will eventually become unmanageable if not taken care of. It is sometimes seen as a way to explain the need for refactoring, i.e. making changes to code that does not change the underlying behaviour or business purpose of the code, to non-technical stakeholders [15].

2.3 The Granularity Problem

How to measure the granularity of microservices is not well-defined nor agreed upon [9]. There are many different approaches, yet it is generally agreed upon that microservices should be loosely coupled and highly coherent. It is also commonly stated that services should have fine-grained interfaces [3]. However, how small or fine-grained to make microservices is an active research topic [3]. Creating a microservice for every new feature was identified as a microservice smell by Taibi and Lenarduzzi, called "Microservice Greedy" because it becomes difficult to maintain systems with many microservices [12]. On the other end of the spectrum, services may become so big that they cannot be called microservices and consequently do not provide any of the advantages promised by microservice architectures. The maximum possible size of a microservice is however not well defined either. In [4] many papers that investigate microservice granularity are discussed, although they are mostly concerned with extracting microservices from monolithic applications.
Furthermore, MSAs evolve, and consequently, they need to be evolvable—evolutionary design is one of the characteristics of microservices Martin Fowler presented when introducing them [2]. Bogner et al [16] state that while participants of their study generally saw their MSAs as evolvable, finding an appropriate service granularity was particularly challenging. They also state that the tool and metric usage of their participants primarily consisted of source-code quality metrics and that no architectural tools or metrics were used.

2.4 Related work

The paper [4] suggests that Domain Driven Design can be used to achieve optimal granularity, where optimal granularity is defined as having a balance between minimum coupling and maximum cohesion values. Although, they still manually create more granular and less granular examples of two services, and then use their method to decide upon which is the best of the manually created alternatives. Other papers such as [17] and [18] have proposed ways to split monolithic services into microservices and also try to find a good balance between coupling and cohesion.

Nevertheless, Carvalho et al. present in [19] that, while practitioners find coupling and cohesion the most useful metrics, they often consider more than these two criteria. The authors mention that despite this, these two metrics are almost exclusively what the existing tooling focuses on and that practitioners often find this insufficient or irrelevant for microservice extraction decisions. Carvalho et al mention an example where practitioners extracted a monolithic system into microservices but noticed that the introduced latency from network calls between these microservices introduced communication overhead, which led to an unacceptable degradation in performance. To resolve this problem, the practitioners introduced a cache layer to the API. The addition of this layer was unintended and caused by the decision of how to split the monolith into microservices, and of course, introduces more future maintenance work. This is an example of *development overhead*.

Shadija, Rezai and Hill [5] investigated how microservice granularity affected performance. They do not mention development overhead, but they found that granularity decision processes are influenced by environmental factors and that purely functional decomposition is not sufficient to determine the appropriate size of a microservice.

Taibi, Lenarduzzi and Pahl [20] carried out a survey of 21 practitioners, investigating the motivations practitioners had for migrating to microservices and the issues they faced when doing so. 15 of these listed maintainability as a motivation for the migration. Maintainability was the most listed motivation, averaging "4-Fundamental to migration" on a 0-4 Likert scale. Nine practitioners also said that it did improve maintainability. This paper suggests that one reason practitioners might choose to adopt microservice architectures is that several large and influential technology companies use them and that they are perceived as "cool", even if practitioners do not truly know the benefits of using microservices, they prefer to follow the mainstream. The authors also mention that one motivation for migrating to microservices is to better support a DevOps methodology.

Contrary to those results, Simhandl, Paulweber and Zdun [21] conducted an eye-tracking study comparing the cognitive effort between maintenance tasks in microservices as compared to monoliths. They found that the practitioners spent considerably more visual effort and time to identify specific features in a microservice architecture, compared to a monolithic system. They also spent more time completing an assigned maintenance task in the microservices architecture. The authors discuss the implications of the results, mentioning that choosing a microservice architecture can increase the maintenance cost of the system and that choosing this approach should be carefully considered—especially for smaller applications.

Vitharana and Daya [22] authored a paper suggesting that—when it comes to

microservices—organisational issues can be a greater problem than technical ones. They mention that several challenges arise from the lack of understanding of how organisational factors affect microservice development. They write primarily about the difficulty of moving from large dependent teams to small independent teams. They also mention that, for successfully using microservices over a long period, it is necessary to have processes to review and refine microservice applications to ensure that they continue to represent only one business capability, as they can drift away from this when modified—however, most companies do not perform such reviews. The authors do not specify *why* this occurs.
The paper [23] describes that Istio, an open-source service mesh, returned to a monolithic architecture after attempting to use microservices. They realized that modularity affects deployment and operations, not just development and maintenance. The developers had significant experience in microservice development from elsewhere and therefore chose this as the initial architecture. They realized that because Istio as a whole is often managed entirely by a single team or individual, keeping components entirely independent adds a lot of operational complexity, adding difficulty for the end-user, i.e. the operators deploying the service mesh. While this paper focuses specifically on the pitfalls of delivering customer-installed software using microservice architecture, it clearly shows that there are aspects of microservices that cause overhead, especially for operations—which is very relevant since one motivation, as described in [20] for using microservices is to provide better DevOps support. The authors go on to state that with the popularity of microservices, it is increasingly likely that microservices will be used in contexts where the costs outweigh the benefits.

3
Methods

The thesis was conducted as an interpretive case study [8], at a large biopharmaceutical company, with multiple large teams working with the development of several software platforms. The data were gathered from two different sources: thematic analysis of interviews and statistical analysis of git repositories. Results were then triangulated based on data from both sources. A visualisation of this method is shown in Figure 3.1.

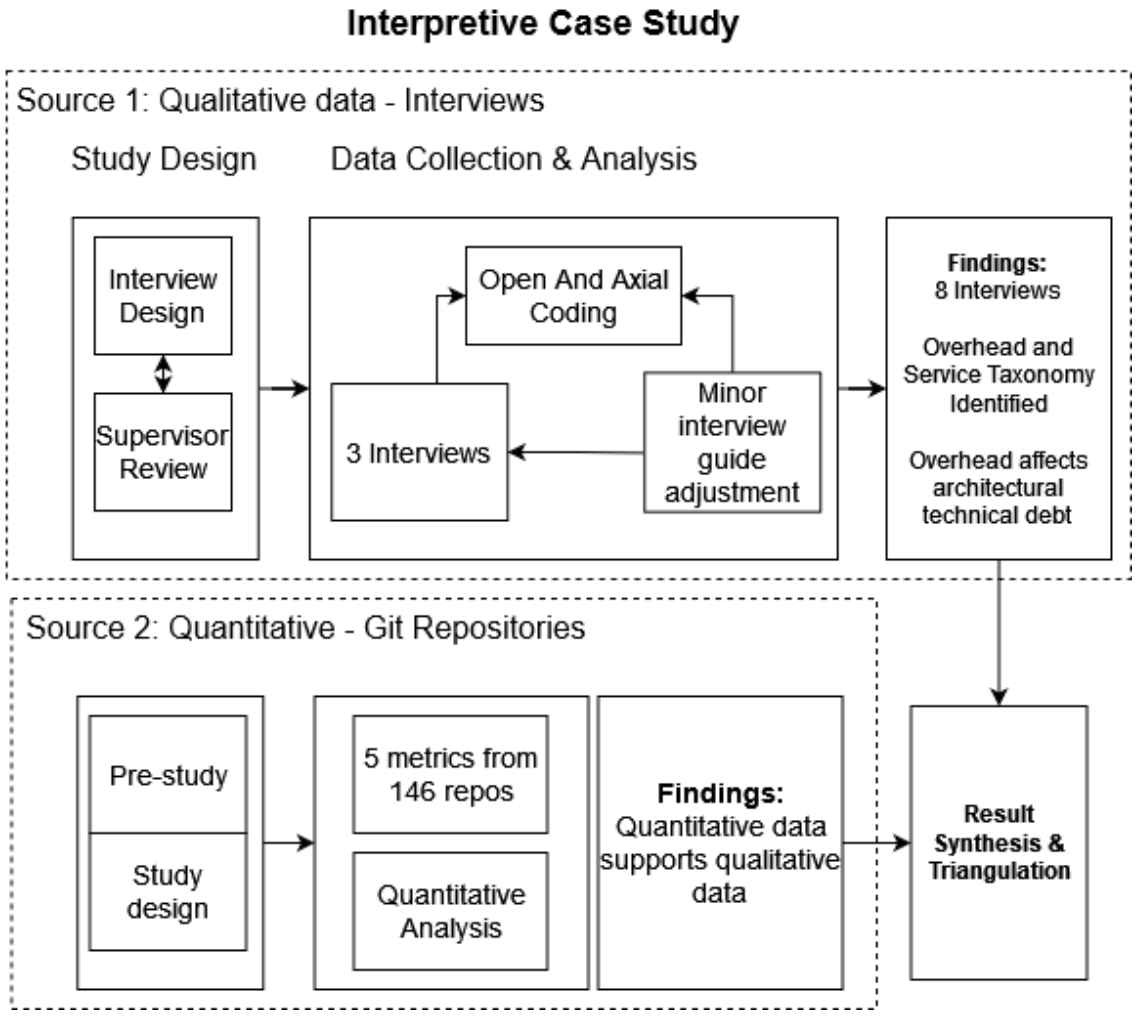

Figure 3.1: A visualisation of the methodology for this thesis

3.1 Qualitative Data Collection and Analysis

The study was conducted as an investigative case study at a large pharmaceutical company with both qualitative and quantitative data analysis. Initially, several practitioners from three scrum teams were interviewed. Subsequently, data were collected from the organisation's git repositories. The qualitative data were iteratively analysed through thematic analysis concurrently with its collection.

3.1.1 Qualitative data collection

Data were initially collected through 8 semi-structured interviews that took place over four weeks. It was done in two iterations of three interviews and a final iteration of 2 interviews. An outline of the finalized interview guide can be seen in Appendix A.1. The people interviewed came primarily from three different scrum teams at the same company. The roles of the people who were interviewed included developers, testers, and architects.

3.1.2 Data coding

As the first step of thematic analysis [24], transcripts from the interviews were coded, i.e. categorised in relation to the research questions. The data was continuously coded concurrently with the collection. This was done through open coding—not using any predefined codes. After this, axial coding, i.e. drawing connections between the codes was performed. This served as the foundation for the lowest-level categories used in the taxonomy created to answer RQ1.

3.1.3 Categorisation Into Themes

After the data were coded, the codes were organised into higher-level themes—something which captures essential aspects of the data in relation to the research questions. This was an iterative process and was reworked several times during the combined data collection and analysis. This served as the foundation for the high-level description of types in the service taxonomy.

3.1.4 Model Creation From Higher-Order Themes and Evaluation

The model created from the themes consisted of an identified taxonomy of service types and overhead, as well as initial results about how practitioners believe the different types of services impact granularity and overhead. It was then evaluated by quantitatively investigating how metrics of the different types differ within the organisation.

3.1.5 Interviewee selection

To select people to interview at the organisation, managers were asked to provide a list of people who were available to be interviewed. The inclusion criteria were that they had been or were working with microservices within the organisation, and had more than 18 months of experience with microservices. The reason 18 months was chosen as the cutoff point is that this was the age of the youngest microservice project, a migration project, at the organisation. If it were excluded it would exclude a significant portion of the organisation's microservices development. The titles and experiences of the interviewees can be seen in Table 3.1.

Table 3.1: Interview participants and their title, software engineering experience and microservice experience. Experience is denoted in years.

Interviewee	Title	SE Experience	Microservices experience
I1	Developer	12	8
I2	Information Engineer	6	4
I3	Technical Business Analyst	1-2	1-2
I4	Solution Architect	20	1-2
I5	Senior Lead Software Engineer	13	7
I6	Quality Assurance Technician	11	8
I7	Business Analyst	7	7
I8	Senior Software Engineer	30	1-2

3.1.6 Quantitative Data Collection

To further investigate the research questions quantitative data was collected through mining git repositories of microservices. The purpose of this was to see how the practitioner's perception matched the qualitative data of the services they had worked with. This data collection occurred just after the initial qualitative data collection was performed. The git repositories were mined for metrics pertaining to granularity, some from theory and others identified through qualitative analysis.

3.1.7 Microservice analysis

If a process, event or situation causes development overhead, it is reasonable that this would increase the time it takes before a commit is made. Therefore, the time between commits was used as a metric for overhead. Of course, there are many things, such as the complexity of the problem which influence the time between commits, but if there are other factors which are correlated with the development overhead it should still be possible to model these.
From the selected repositories, the following data were gathered:

1. Number of non-configuration files
2. Number of configuration files
3. Number of unique contributors
4. Basic service type information (*api, data integration, business logic, other*)
5. Time between commits

Since communication between services is described as a factor which increases when services are split and such communication is often configured with configuration files, it seems that the proportion of configuration files in a repository might be a factor that influences development. This was the reason the number of configuration files and the number of non-configuration files were extracted. The raw counts are also useful since it might be expected that a larger repository is more complex and this is something which would impact the time between each commit. For the same reason, the number of unique contributors is also extracted — having more contributors to a repository should reasonably reduce the time between commits to this repository. If there are more unique contributors to a repository then it is a clear indication that the service has grown in complexity.
Basic type information was also extracted. The type information was simply extracted by looking for a specific suffix in the repository name more than half the data did not have these and were therefore classified as "other".
The definition of "configuration file" in this context is any file with the following file extensions: *yml, yaml, json, cong, cnf, cfg, ini, cf, config, properties, Dockerfile*. Also, several files were ignored because they are neither configuration files nor source code files. These files were files like .gitignore files, which are often the same across all repositories using the same technology at an organisation, or they were documentation like README files. Specifically, the ignored file extensions were: *.gitignore, .helmignore, mvnw, .md, .npmignore, .editorconfig, .gitkeep, browserslist, .dockerignore, npmrc*. The metrics are extracted for each commit of every repository so that the metrics are tracked over the evolution of the microservice.
The metrics from each selected repository were extracted and then inspected to look for patterns.

3.1.8 Microservice selection

To select microservices to analyse, the organisation's definition of microservices was used. They have several platforms that exclusively contain microservice projects, so these formed the initial selection of microservices and totalled 167. Some services were then excluded if they were either empty, only contained schema definitions, or if they only contained documentation. They were also removed if they were services only meant as a proof of concept and not actually in use. The services were automatically removed if they contained fewer than 5 commits, or if the repository name contained "docs", "checkstyle" or "poc" because they were likely to be one of these types of services, but determining which services met the exclusion criteria was also done manually after these automatic exclusions. After these services had been excluded 146 services remained. An overview of the microservice repositories is displayed in Table 3.2.

Table 3.2: Overview of the 110 mined repositories. Mined 2023-04-25.

Metric	Mean	Median	Min	Max
Commits	166.4	55	1	2328
Files	55	60	1	3067
Contributors	16	13	1	85
Age (Days)	613	509	8	1639

4
Results

This section provides a taxonomy of the identified types of services at the organisation, discusses the different ways practitioners describe granularity, and provides a taxonomy of different types of overhead. Then it relates these to one another. Some key results are that, in the practitioner's perspective:

1. The service type will impact how the granularity of the microservice changes over time.
2. There is a small subset of larger microservices that grow far more than others in the microservice architecture.
3. Most microservices tend to stay the same size over time.

It is found that the quantitative data supports these key qualitative results.

4.1 Excluded metrics

It was found that the time between commits and the proportion of configuration files in relation to non-configuration was not useful for finding patterns in the data. Albeit, the total file count along with unique contributors for a repository was a useful metric for size and provided some insight.

4.2 Service taxonomy

The interviewed practitioners described many types of services and noted some differences in working with them. The services categories are presented here, first through a broad description of what the practitioners described, then how this aligns with the quantitative data. A visualisation of the types can be seen in Fig. 4.1. Note that not all of these categories of services are mutually exclusive.

4.2.1 Scope

The practitioners described differences in working with internal or external services, referring to things that are internal to their platform or organisation and that do not need to interact with third-party services.

- **Internal** services require less communication with other teams
- **External** services require more communication with other teams

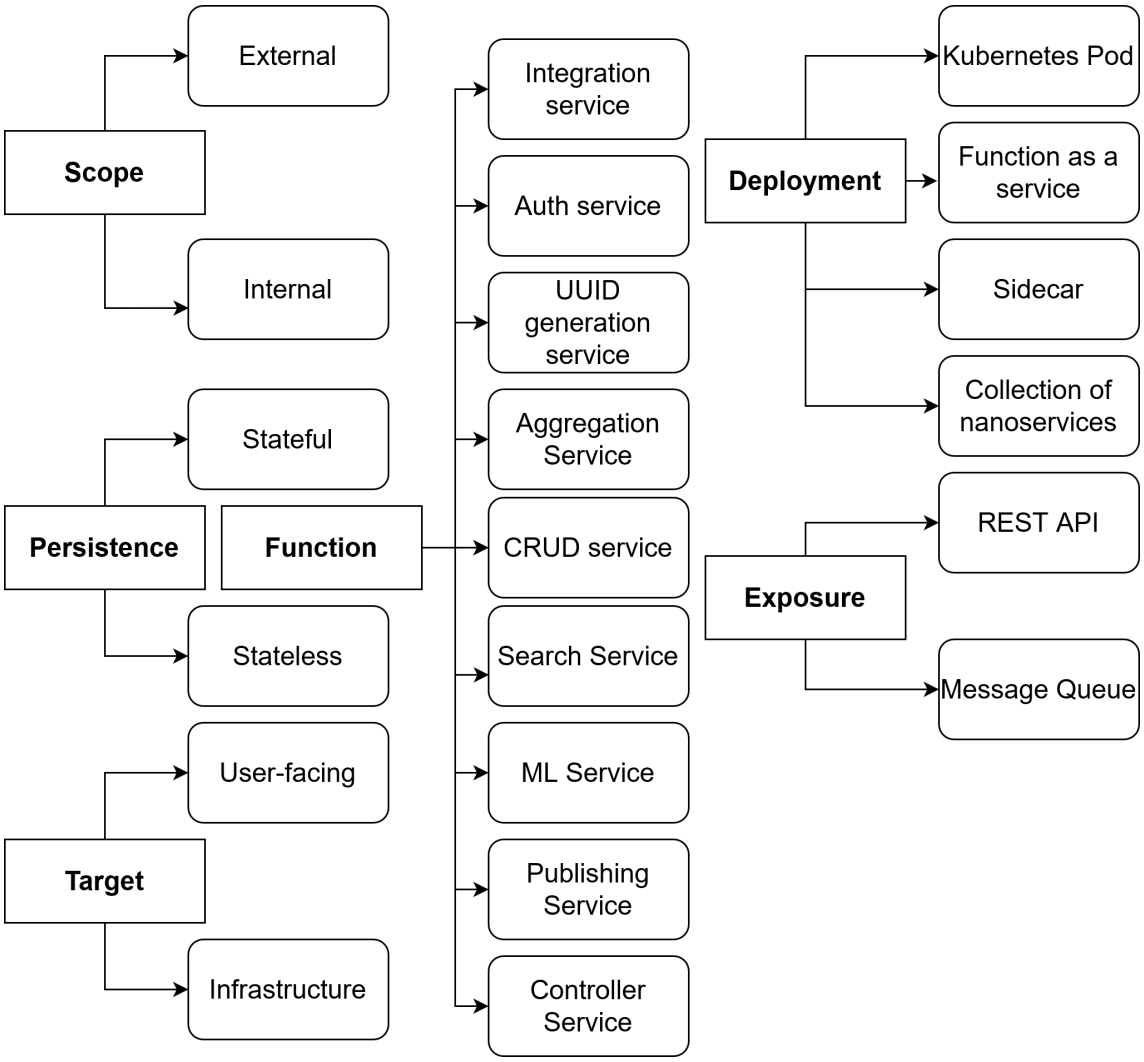

Figure 4.1: The categories of different services described by the practitioners

4.2.2 Persistence

The interviewees mention the need for persistence as a significant distinguishing factor between services.

- **Stateless** services are services that do not need to persist data. They can for example be services that read data from one or several message queues, perform some transformations and send it to one or more message queues. According to the practitioners, these tend to be very simple.
- **Stateful** services are services that persist data in some form, such as through a database. These services can be simple but tend to be larger than stateless ones.

4.2.3 Target

While all services contribute to the functionality for the end-user in some way, some services provide direct functionality for users whereas others services provide the

necessary infrastructure to fulfil that functionality.

- **User-facing** services are services that directly provide an interface to do things that are directly useful for the user. These services change often, because users make demands on them, evolving the requirements. This can either be an end-user, or a UI developer requesting data in some different format because performing data transformations client-side is inefficient.
- **Infrastructure** services change less frequently because often the infrastructure required (such as data pipelines) to make changes that meet new user requirements already exists, it just has not been done yet.

4.2.4 Function

Services perform a plethora of tasks but they can be categorized into some high-level functionality types.

- **Integration** services are simple services that form data pipelines which send data from one or many places to one or many other places.
- **Auth** services, short for Authentication & Authorization, are services responsible for managing identity and permission for the services. Since all other services need these, they tend to be very large.
- **UUID generation** services are services that provide unique identifiers based on business rules, for the platform or organization. They end up being very simple.
- **Aggregation** services are services that aggregate data from multiple different sources and write the result to a data store or message queue.
- **CRUD** services are services that essentially only provide an interface to a database.
- **Search** services are services that provide some search interface over data.
- **ML** services are services that provide an interface to a deployed machine learning model. Usually, they just serve a containerized model, so they are very simple.
- **Publishing** services are services that convert message queues into a format that a relational database expects, to easily search through or log transaction data.
- **Controller** services are services that decide which other services to call or which message queue to write to depending on some business logic. They tend to be very complex.

4.2.5 Deployment

The practitioners described different ways they have deployed their service. While the different deployment types do not necessarily contribute to the complexity or lines of code of the service, they do impact the work associated with deploying them, as well as the granularity of the interface of the service. How it is deployed is very relevant for how much work that is not programming is involved in the microservice development, because these different deployment methods entail other architectural

considerations, but they also at the organisational level entail different processes, notably, how access management is handled.

- **Kubernetes pods** are the smallest compute unit deployable on a Kubernetes cluster and is how most of the services at the organisation are deployed, one microservice per pod.
- Some services are deployed as a **Function as a service**. These have a very simple interface consisting of a single function, but they can be very complex.
- **Sidecar** Some services are deployed using the side-car pattern and are deployed alongside another microservice in the same Kubernetes pod.
- **Collection of nanoservices**
  Sometimes several different functions as a service are seen as one microservice, serving one domain.

4.2.6 Exposure

Another difference between services is how they interact with the rest of the IT landscape. At the organisation, it is done exclusively in one of two ways, either through a REST API or through a message queue.

- A **REST API** lets users or other services call the service and receive some kind of response or initiate a process. Tend to be larger since they need to maintain an API, verify the authentication of the caller, etc.
- If the service uses a **message queue**, it is instead only interacted with asynchronously, where it polls messages from a message queue continuously. Tends to be simpler, but the business logic can still be very complex.

4.3 Development overhead taxonomy

The practitioners also describe many different types of overhead they have experienced, see Fig 4.2. Essentially all developers defined development overhead as anything that impeded them from making changes or deploying them.

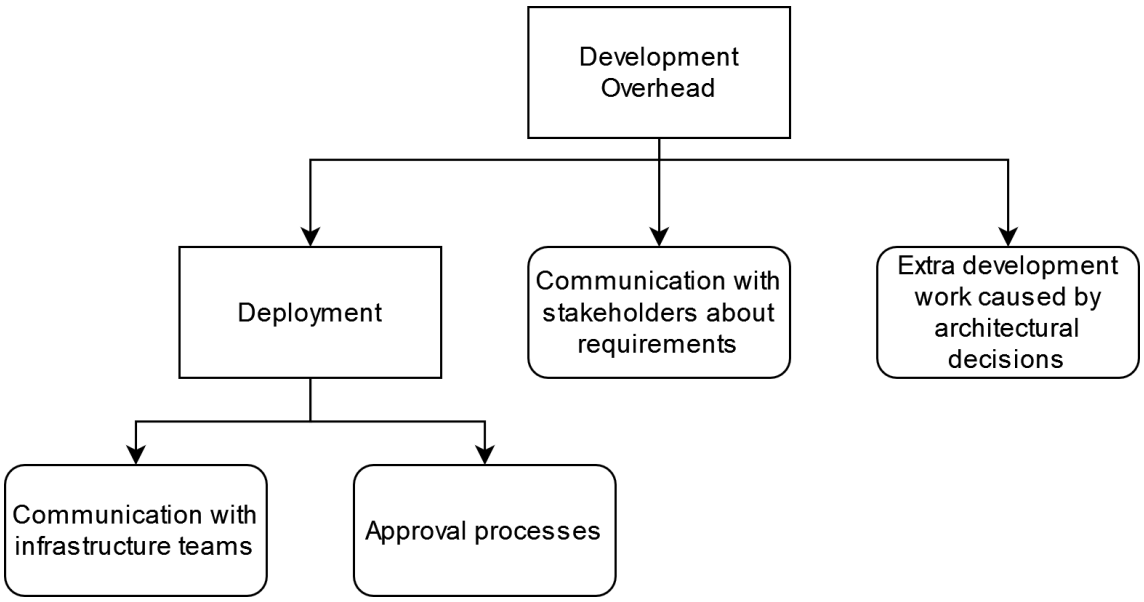

Figure 4.2: A taxonomy with different categories of overhead

4.3.1 Deployment

Most of the interviewed developers primarily talk about overhead as challenges with deployment, getting the required permissions for their services to access other databases or other services etc, and the approval process to get the services approved. Development overhead is commonly also described as communication with other teams.

> Development overhead, how would I define it? I suppose anything that impedes your path to making changes or getting them deployed **I1**

- **Communication with infrastructure teams**. When deploying services, the developer often does not have control over test, preproduction or production environments, and often they lack permissions to directly configure the CI/CD pipeline or secrets key vault being used. Developers give examples of having to wait for teams responsible for these aspects, such as updating a secret or configuring a database in a test environment as something they frequently have to wait for. Often they tell these teams exactly what to do, but are not permitted to do it themselves.
- **Approval processes**. Performing the initial setup and getting everything ready to deploy a microservice is described by some practitioners as the most time-consuming part of the development effort. One practitioner describes

building the microservices as very easy compared to getting permission to build and deploy a specific thing with the required access.

4.3.2 Communication with stakeholders about requirements

Essentially every interviewee stated that they find the requirements engineering to be the hardest part of microservice development, even the developers, who mark the importance of well-defined requirements. They state that creating something based on incorrect assumptions is something that happens fairly frequently and that they need to thoroughly communicate with the Business Analyst / Product owner to understand the requirements. This is further exacerbated in microservice development since sensibly splitting the microservices requires having well-defined domains.

4.3.3 Extra development work caused by architectural decisions

When services were being migrated and were newly designed, architects did concern themselves with granularity and purposefully made decisions to reduce overhead resulting from too small services. For example, one team had a validation service doing four different types of validations. Some team members believed it could be split up, but the architect argued that splitting it up would increase the communication overhead between the services since there is no point in continuing if any of these four validations fail. They also stated that the complexity of maintaining four different systems would be far greater than keeping it as a single system and that they wanted to keep it as simple as possible.

> [if you split it up] you're really just going to add to the communication impact of this entire solution, there would be additional communication that we'd need, additional compensation that we'd need, I prefer to just keep it as simple as possible I think...
> **I4**

Another architect (**I5**) stated they in general wanted to keep them larger because they had had a lot of problems with communication between services before.

4.4 Perception of granularity

There are many ways to measure granularity, and practitioners state many different ways they see it. Some, such as complexity, are well-studied in the existing literature, and this is one metric that is mentioned by the practitioners when asked which services they have worked with that are the largest or smallest. However, they also describe how they see some services as very large or very small because of the size of the service interface. A service with many endpoints can be seen as smaller than a service with far more lines of code and more complexity, but a lower number of endpoints.

4.5 Perceived impact of overhead

Some developers talk about how the overhead in deploying services causes them to make pragmatic granularity decisions, they get new requirements and have deadlines to meet, and therefore feel that they need to do what is the easiest, and sometimes granularity decisions are not considered.

> "[sometimes] the overhead of trying to create a new service and get it deployed etc, means it's easier to put it in the existing one.
> **I1**

They state that this primarily happens as microservices evolve over time and new requirements are received. **I2** describes that it's easy to lose track of what a service does over time and that you get feature requests constantly from users, causing you to add new features to the services without thinking whether they belong in that service or not.

4.6 Relation between the service taxonomy, overhead taxonomy, and granularity

One practitioner states that *user-facing* services tend to change, because of new requirements, and other practitioners agreed with that sentiment. The practitioners think that this is why they tend to get larger than other services because of constant changes causing pragmatic granularity decisions.

> Well, some services are more fixed [...] we don't really [do a lot of new development on them] but [user-facing services] they tend to have, uh, more functionality coming into it: "Now I want to be able to add this now. Now I want to be able to add that..."
> **I7**

And other practitioners agree with the sentiment. They state that *integration* services and *infrastructure-oriented* services tend to remain small and simple, but that everything else grows.

> You don't really add so much logic to [communication services], they're kind of going to stay the same for a long time they're not going to change in that sense [...] but other, the heavier services, they don't tend to stay the same size.
> **I5**

While the practitioners do not directly state it, they also discuss that interacting with teams about databases can be frustrating and create additional work, so stateful services probably entail more work than stateless ones. In the same vein, services that only interact with internal services require less work than those that interact with external services.

4.7 Quantitative analysis

The key qualitative results, as discussed in the previous section, were that, in the practitioner's view, the following is true:

1. The service type will impact how the granularity of the microservice changes over time.
2. Most microservices tend to stay the same size over time.
3. A small subset of larger, so-called "heavier", microservices grows far more than the others in the microservice architecture

Quantitative analysis that supports these results is presented in this chapter.

4.7.1 Does the service type impact how the granularity of the microservice changes over time?

It can be seen in Figure 4.3 that there are significant differences in service growth between various basic types. The boxplots show the distribution of the file difference between the commit at the midpoint in the repository's history and at the last commit, for each service category. The indicated outliers are repositories below the 2.5th percentile or above the 97.5th percentile, i.e. the 95 % central range. There is also a further outlier for the unspecified types, which has grown by more than 1000 and has been excluded from the figure for the sake of clarity. The reason this method was chosen to identify outliers is that the sample size is small and the data is right-skewed, causing other methods, such as using the interquartile range, to result in a significant number of outliers. The same method will be used for all further boxplots presented in this thesis.

This difference between the types is further highlighted in Figure 4.5 which shows that the growth in the number of contributors varies considerably between the types. The figure contains boxplots that show the distribution of the difference between unique contributors between the middle and the last commit, for different types of services. The service type that grows the most is functional services, where more than half grow by more than 15 unique contributors between the middle-point of the repository's history and the latest commit. The service type that grows the least is the "Integration" type where all repositories have fewer than 10 additional unique contributors at the last commit compared to the middle commit. This can be seen more clearly in Figure 4.4.

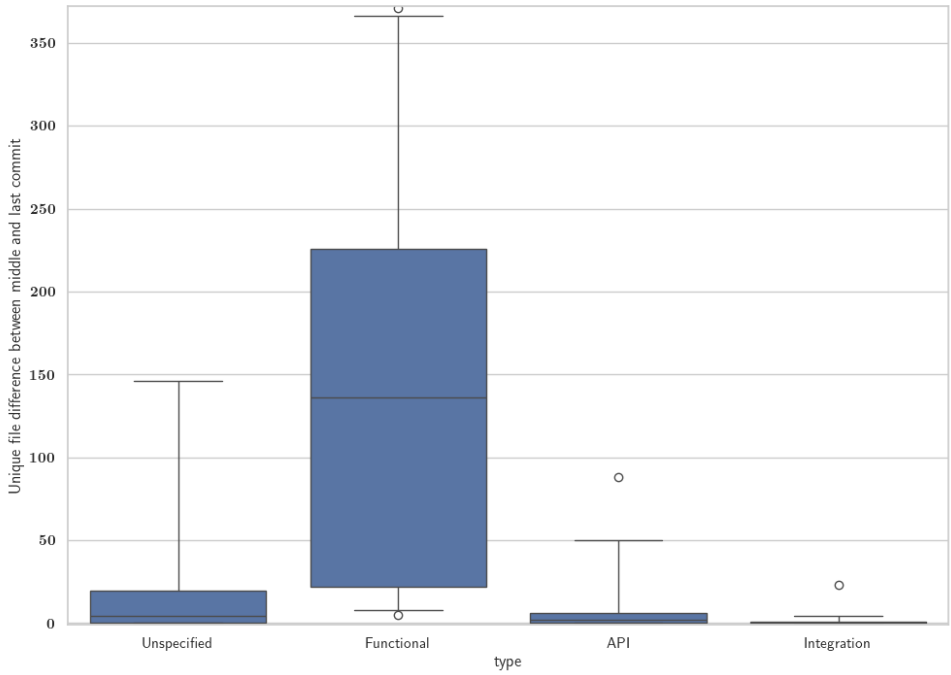

Figure 4.3: Boxplots showing the distribution of the file difference between the mid-point and last commit of a repository's history.

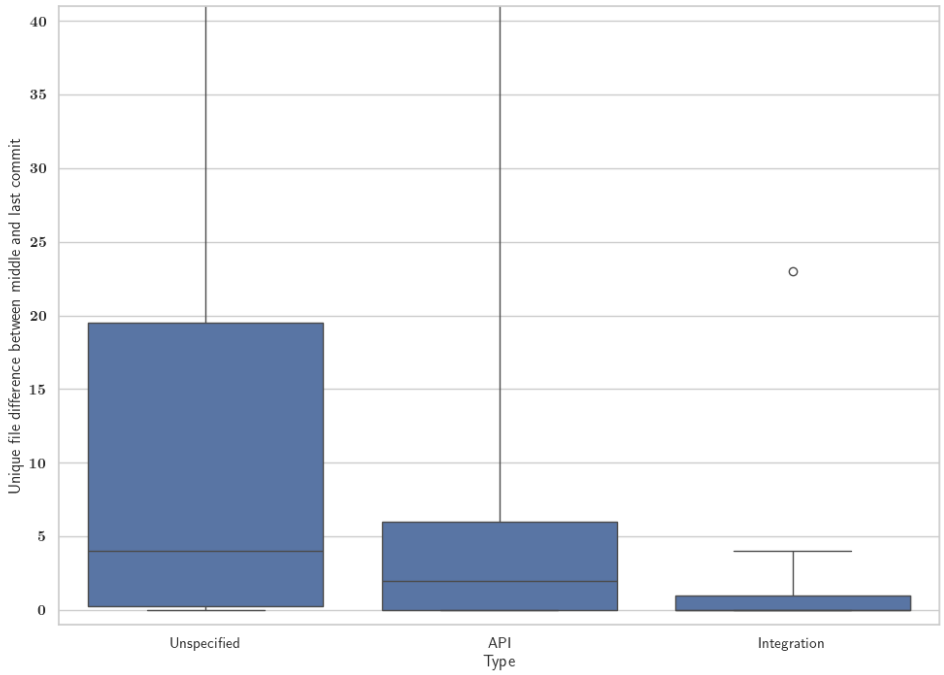

Figure 4.4: Boxplots showing the distribution of the file difference between the midpoint and last commit of a repository's history, excluding functional types.

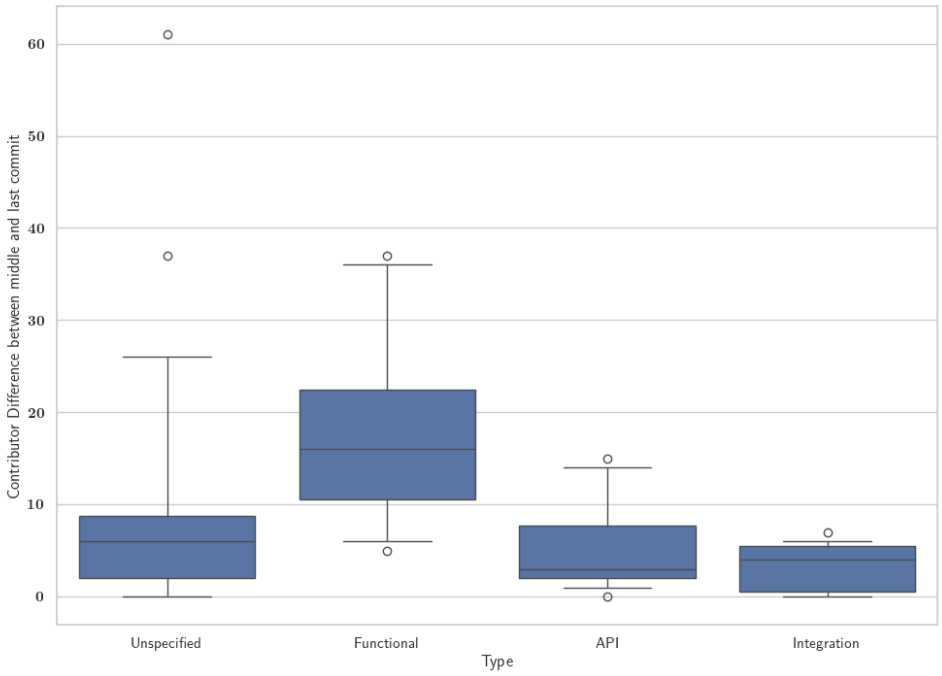

Figure 4.5: Boxplots showing the distribution of the difference between unique contributors between the middle and the last commit, for different types of services.

4.7.2 Do most services tend to stay relatively the same size, while only some change over time?

To investigate this, the file count of the commit at the midpoint in a repository's history is compared to the file count at the last commit in the repository's history. Figure 4.6 contains two bar plots, one shows how many repositories had which number of files at the commit in the middle of the repository's history, and the other the repository count for the same at the last commit. It can be seen that the distribution remains approximately the same between the midpoint of the repository and the latest commit, and therefore be concluded that most services remain approximately the same size, but it can also be seen that there are some outliers. There were 92 out of 110 repositories with fewer than 200 files (84 %) at the repository's midpoint, and 90 of out 110 repositories with fewer than 200 files (82 %) at the last commit. There were also 75 out of 110 repositories with fewer than 100 files (68 %) at the repository's midpoint and 74 out of 110 repositories with fewer than 100 files (67 %) at the last commit.

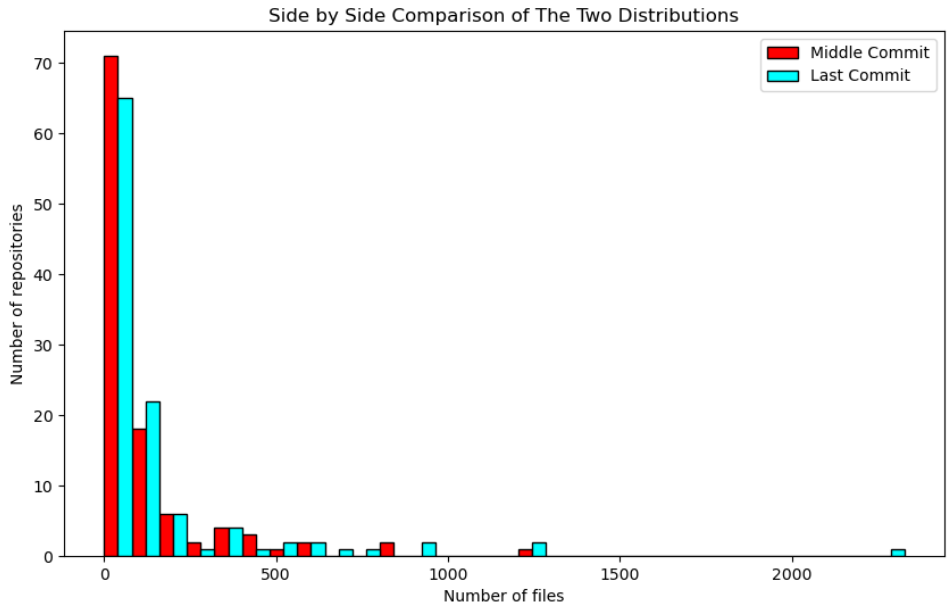

Figure 4.6: Overlayed plots of files at the midpoint of the repository and at the last commit.

4.7.3 Do "Heavier" services grow more than others?

To investigate this, the number of files and contributors at the middle point of the repository's history is compared to the file count and unique contributor count at the latest commit made to the repository. The reason that the file count and contributor count at the middle commit are used for the comparison is to avoid including the initial development effort and to do the comparison between more established repositories.

It can be seen in Figure 4.7 and Figure 4.8 that the mean of the difference, for both file counts and unique contributors, between the middle and the last commit, grows with the size of the repository. The X-axis of Figure 4.7 shows the number of unique contributors at the midpoint point of the repository's history. The Y-axis shows the difference between the number of unique contributors at the latest commit and the middle commit. The X-axis of Figure 4.8 shows the number of files at the middle point of the repository's history, while the Y-axis shows the difference in total file counts between the middle and last commit.
There is also a blue regression line plotted through both plots to highlight the mean, but it is not expected that this can be used for extrapolation, since surely the practitioners would at some point realize that the service is too large and that it needs to be split. Albeit, that should arguably have been done for some of them already, since even when accounting for new members joining the team, people leaving the team, and people moving between teams, having more than 40 unique contributors to a microservice repository, which six repositories have grown to have by the latest commit, is an indication that is unlikely to be manageable by a single team. The two figures also show that it is primarily functional services that are already very large at the middle commit and that it is often these that grow substantially.

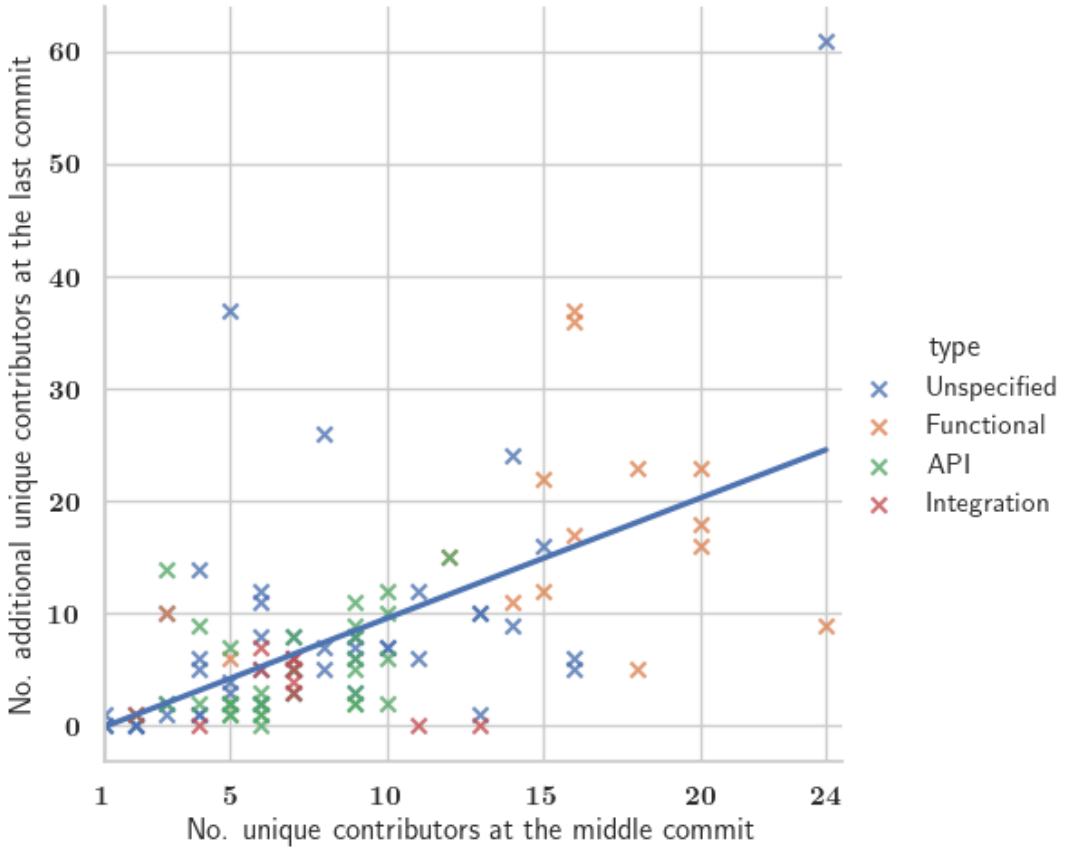

Figure 4.7: Number of additional unique contributors at the last commit over the number of unique contributors at the middle commit. The blue line displays the mean.

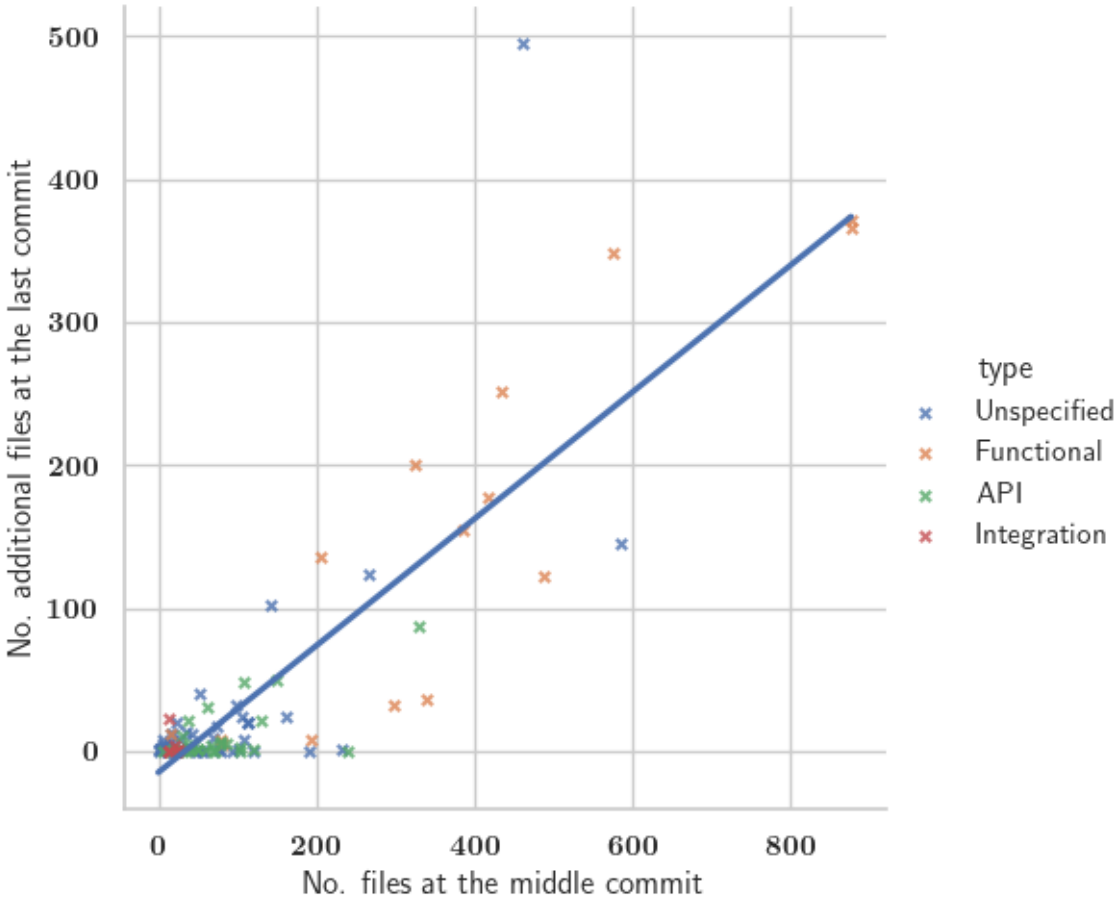

Figure 4.8: Scatterplot showing that the larger a service is at its midpoint the more likely it is to grow even further.

5
Discussion

It is clear from the results that the practitioners think that there is a relationship between service type, development overhead, and microservice granularity. The qualitative data also supports this. Development overhead was described primarily as extra work with deployment and changing requirements. The most interesting finding is that the size of certain types of microservices grow unintentionally due to this development overhead.
Notably, there is a small subset of services that grow larger than intended, while most services do stay roughly the same size. The services that grow are functional, mostly user-facing services and services that have already grown large, and this occurs primarily due to more frequent updates of the requirements. The practitioners have also discussed that services often grow larger than intended for pragmatic purposes, such as tight deadlines and tedious deployment processes, where increased granularity is a form of accepted technical debt. That tight deadlines cause technical debt is well described in the literature, with one study [25] showing that it is the most cited cause for it by both junior and experienced practitioners. Novel for this study is that this can take the form of not creating a new microservice, even if the practitioner thinks that should be done. That study also found that experienced practitioners consider inappropriate planning and ineffective project management to be the second and third largest causes of technical debt.
The novelty of this thesis's findings is that microservice granularity is a factor that is considered by architects at design time but that the size can deviate from the design in undesirable ways during the evolution of the microservice system, especially for certain types of services — primarily functional, user-facing services — due to these services being particularly prone to technical debt.

5.1 Implications

There are two interesting implications of the findings—the first is that it is important to make it as easy as possible for the developer to put a new feature in a new service if they think that is appropriate. If it is difficult, it lowers the threshold for accepting technical debt and leads to the perceived poorer alternative being intentionally chosen in the interest of time.
Another interesting finding is that it would make sense to be extra thorough when planning certain types of services, those particularly prone to change, and for project management to be cognizant of the effect of agreeing to or not properly prioritizing new requests for changes. When building a service with a lower risk of unexpected

growth, it is less important. For example, there is an architectural pattern called backend-for-frontend, where a microservice is built which receives data from a standardised API, and provides a customized API to a specific frontend in a way that makes it simple to display without further transformation, to enable the frontend to not include complicated business logic. Such a service is an example of a user-facing service that is very prone to change because the users interacting with it will have new requests as they use it. The standardised API itself is of course less prone to change.

5.2 Theoretical relevance

Prominent literature such as [20] suggests that maintainability is a primary consideration for and benefit from microservice migrations, while other literature such as [21] provides evidence that microservices can result in increased maintenance effort and that microservice adoption should be carefully considered. There is also literature [22] suggesting that organisational factors have an impact on the success of microservice adoption. It has also been shown that using microservices can increase the complexity of operating the system [23]. Adopting microservices is viewed extremely positively in [20], and building microservices because "everyone else is doing it" is listed as a motivation. The papers [23, 21, 22] suggest that this viewpoint is inappropriate and that one should only decide to use a microservice architecture after careful consideration to ascertain that the benefits will truly outweigh the costs. The results of this thesis provide further evidence of this by showing that there can be additional costs in the form of development overhead and that changes to organisational processes may be required.
While not directly supporting the findings of [21], the findings of this thesis support the notion that there are other factors which can increase developer's efforts in the development of microservices and that these also need to be considered. The results show that there can be further detrimental aspects of microservices and that increased maintainability as suggested by [20] is no guarantee—especially if organisational processes do not support microservice-oriented development. These factors are primarily organisational, such as the ease of deployment, which can decrease maintainability—although this is not apparent through conventional metrics, as it does not concern the source code itself, but rather the difficulty of getting a new change to a production environment. The paper [22] also proposes that microservices need to be periodically reviewed to be kept at an appropriate size, and the results of this thesis support this and show that some services do grow too large. The findings also suggest that the frequency of these checks could vary for different service types since the risk of growing too large differs between the types. There could be more frequent checks for larger services that have frequently changing requirements, and less frequent checks for smaller services with more stable requirements.
The literature focused on granularity and microservice architectural decisions, such as [4, 5, 17, 18] generally consider coupling and cohesion and its effect on performance and scalability. The results of this thesis show that it would make sense for this organisation to consider other factors too, and also consider the process around the development and deployment of microservices, and how these can affect the

granularity. [19] also indicates that practitioners do want to consider other factors, but the tooling that exists to do so is inadequate. The result of this study suggests that there are factors that affect microservice granularity and especially its evolution that is beyond what developer or architect tooling could accomplish because it is dependent on organisational processes themselves. It has been described by [25] that organizational culture, tight deadlines and ineffective project management are the perceived main causes of technical debt, and that appears to be the case at this organisation too. Nevertheless, this study identifies that this in particular affects microservice granularity, especially for specific service types, and that is worthy of further study.
This thesis supports literature suggesting that organisational factors can be as important as technical ones concerning microservice architecture systems and supports research showing that there are factors which are not immediately apparent that can impact the maintainability and evolution of microservice architectures. It also suggests that surveys showing that maintainability is the main benefit of adopting microservice architectures might not convey the whole picture and that this is no guarantee. It is possible that migrating to microservices primarily benefits systems that are already difficult to maintain, and that more maintainable monoliths of a smaller size might not benefit from a migration. As discussed, microservices do have both advantages and disadvantages. Those disadvantages could potentially outweigh the advantages. The results show that there are further considerations to be had and organisational changes that need to be made to not exacerbate those disadvantages. This implies that there is an even greater reason than previously described to be very cautious about choosing to use a microservice architecture, especially for smaller systems, and to ensure that the advantages weigh up for the disadvantages.

5.3 Relation to research questions

The first research question, what categories of services exist within the organisation, was partly answered by providing a taxonomy of services based on a thematic analysis of the interviews with practitioners at the organisation. Five key dimensions to separate services by were identified: scope, persistence, target, deployment and exposure. The key result is that practitioners associate different levels of development overhead with different service types, a lot of which comes from the different rates at which the requirements for these service types change. The interviewees also provided what the positive and negative aspects of certain types of services were, and which services tend to grow more than others. This was also supported by the quantitative data, showing through descriptive statistics that services can very easily be categorized into several types and that functional services have a tendency to grow more than others.
The second research question, what practitioners think development overhead is, and where it comes from, was also answered through thematic analysis of the interviews—There is a perception among members of the organisation that smaller microservices result in more development overhead, and that some of this overhead comes in additional work writing deployment configurations and such. They also

specified that they perceived other problems arising from microservices, such as increased communication with stakeholders due to the complexity of requirements engineering due to the need to split the business into well-designed domains. Some also noted that they needed to have more communication with infrastructure teams since there was more to deploy.
The third research question, what relation there is between development overhead and microservice granularity was answered by combining qualitative and quantitative analysis—Several practitioners said that they had intentionally made suboptimal granularity choices, or not thought about it at all, due to rapidly changing requirements of functional, especially user-facing, services, combined with the additional challenge of splitting the service. While those challenges can be architectural, they are often, as discussed by practitioners, caused by organisational processes. This has resulted in these services drifting away from the initial architectural granularity decision and becoming much larger than intended. This was not the case for all such services, but they are much more prone to it. Quantitative data shows that functional, larger services grew more depending on how large they already were at the mid-point of their history, both in terms of the number of files and also number of contributors. It was discussed that this can be seen as a form of architectural technical debt caused in part by development overhead.

5.4 Threats to validity

As described in the method, certain file extensions were included as configuration files, while some were ignored. While an attempt was made to make it as accurate as possible, it is still unlikely that every configuration file was caught and that all files, like documentation, that ideally should have been ignored were ignored. This is because it wasn't feasible to go through every repository manually.
Also, there was the risk of subjectivity in interviews. People are prone to subjectivity and bias and a common theme was to describe what one had personally been involved in creating as good. This threat was mitigated by comparing data about the organisation from multiple sources and triangulating the results.
Another threat is researcher bias. A large part of the thesis consists of qualitative analysis; therefore, interpretation can be subjective. This is mitigated by the fact that the qualitative analysis only serves to be broad and descriptive and that the results are triangulated from multiple sources.
Yet another threat was selection bias. The people who were available to be interviewed were communicated through managers at the company, and they could theoretically select people to impact my results in a way they might perceive as beneficial. This was mitigated by picking interviewees suggested by different managers, and not sharing the questions that were going to be asked with the managers.

5.4.1 Generalizability

The study was conducted as an interpretive case study, with a single case. However, it was conducted at a large pharmaceutical company with a very large in-house software development department. Interviews were also carried out by practitioners with different roles, belonging to different teams working independently from one another with multiple different platforms. The practitioners also described different organisational processes, for example, the ease of deployment differed between the teams.
Of course, it is also possible that since they belong to the same overarching organisation, this influences the organisational processes the developers have to directly deal with—even if they differ between teams—in a way which is specific for this organisation. While it cannot be proven that the results generalize there are good indications that they likely do.

6
Conclusion

A case study was conducted at a large pharmaceutical company to investigate how they categorize their services, what they perceive as "development overhead" and what relation they think there is between development overhead and microservice granularity. It was found that they categorize their services primarily by whether the scope is only for internal or also external stakeholders, how persistence is handled, whether the service is infrastructure or user-facing, how the service is deployed and how the service is exposed to the rest of the IT landscape. It was found that the practitioners think that the development overhead is primarily deployment work and necessary communication with other teams. They also believe that the service type does matter in impacting this overhead, and this is also supported by the qualitative data. It was also discussed how this relates to other research on the topic, highlighting that some microservice architectures have failed because of unforeseen consequences, especially concerning deployment, and that organisational factors are important. This thesis supports those findings, identifying even more potential problems of microservice architectures, the importance of organisational processes, and suggesting that there's further cause to be cautious before choosing to use a microservice architecture. Future research could be carried out to find how the size of a microservice changes for specific service types depending on the organisational processes. Specifically, it could be studied how the microservice granularity evolution differs depending on the ease of deployment.

Bibliography

[1] A. Cockroft, "Migrating to cloud native with microservices," December 2014.

[2] M. Fowler and J. Lewis, "Microservices," March 2014.

[3] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, pp. 301–310, 7 2017.

[4] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?," *IEEE Access*, vol. 9, 2021.

[5] D. Shadija, M. Rezai, and R. Hill, "Microservices: Granularity vs. performance," 2017.

[6] H. M. Ayas, P. Leitner, and R. Hebig, "Facing the giant: A grounded theory study of decision-making in microservices migrations," 2021.

[7] V. Bushong, A. Abdelfattah, A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Černý, K. Frajták, P. Tisnovsky, and M. Bures, "On microservice analysis and architecture evolution: A systematic mapping study," *Applied Sciences*, vol. 11, p. 7856, 08 2021.

[8] P. Ralph, "Toward methodological guidelines for process theories and taxonomies in software engineering," *IEEE Transactions on Software Engineering*, vol. 45, 2019.

[9] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, "Defining and measuring microservice granularity—a literature overview," *PeerJ Computer Science*, vol. 7, 2021.

[10] T. Wagner, "Microservices without the servers," 9 2015.

[11] E. Reinhold, "Lessons learned on uber's journey into microservices," 6 2016.

[12] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, 2018.

[13] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, 2022.

[14] W. Cunningham, "Debt metaphor."

[15] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[16] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Assuring the evolvability of microservices: Insights into industry practices and challenges," 2019.

[17] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," vol. 9846 LNCS, 2016.

[18] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," 2017.

[19] L. Carvalho, A. Garcia, W. K. Assuncao, R. D. Mello, and M. J. D. Lima, "Analysis of the criteria adopted in industry to extract microservices," 2019.

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

[21] G. Simhandl, P. Paulweber, and U. Zdun, "Developer's cognitive effort maintaining monoliths vs. microservices - an eye-tracking study," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 339–348, 2023.

[22] P. Vitharana and S. A. Daya, "Adopting and sustaining microservice-based software development," *Commun. ACM*, vol. 67, p. 34–41, jul 2024.

[23] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 5, pp. 17–22, 2021.

[24] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, 2006.

[25] S. Freire, N. Rios, B. Perez, C. Castellanos, D. Correal, R. Ramac, V. Mandic, N. Tausan, G. Lopez, A. Pacheco, D. Falessi, M. Mendonca, C. Izurieta, C. Seaman, and R. Spinola, "How experience impacts practitioners' perception of causes and effects of technical debt," 2021.

A

Appendix 1

A.1 Interview Guide

1. What is your educational background, how long have you worked at your company, how long have you worked with software engineering, and how long have you worked with miroservices?
2. How would you define a microservice, and what is the biggest and smallest services you've worked with respectively?
3. Do you think any of those microservices became too big or too small, and if so, why? Also, if they did — Why?
4. What services have you worked with that are between these two extremes? What is their domain — what types of services are they? Are certain types of services more often a specific size?
5. Have you done anything in your work recently that you found either tedious, annoying or repetitive?
6. If you have done anything tedious, annoying or repetitive in your work? What was it? Why? Could it have been prevented? If you haven't, why do you think that is?
7. Have you been blocked from doing your work lately by anything? Why? What could have been done to prevent it?
8. What do you find to be the most time consuming task when developing a microservice? Why? Can you think of any way to speed it up?
9. How long does it take for things to move from requirements to deployed in a development environment, and then to production? What practices and proceedures are in place for that?
10. Are you involved in deciding whether new requirements should be included in an existing service, or if a new one should be created? If you are, what influences those decisions? If you aren't, who is?
11. If you could do things your way and do anything you wanted, what would help you develop microservices? In particular, what would help you decide on the granularity of the a microservice?