jamk

Comparison of Monolithic, Micro-service, and Cloud development

Toni Pikkumäki

Master's thesis
April 2023
Master's Degree Programme in Information Technology
Full Stack Software Development

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Pikkumäki, Toni

Comparison of Monolithic, Microservice, and Cloud development

Jyväskylä: JAMK University of Applied Sciences, April 2023, 106 pages.

Master's Degree Programme in Information Technology. Full Stack Software Development. Master's Thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Modern software development offers a wide variety of choices. There is no one technology above all others; instead, all have their advantages and shortcomings. One way to decide which technology to select when developing software is to define its main use case and perform a comparison between the target technologies.

Three technologies with different software architectural models were selected for this investigation: .NET Monolithic, Java Spring Boot Microservice, and AWS Cloud. A different development language and software architecture was chosen for each, although the focus was on the latter. The main goal was to find the strengths and weaknesses of each selection and how they compare with each other. The research focused on developing applications' core logic, omitting testing, and DevOps aspects of software development.
A prototype was built with each of the selected technologies, with the goal of making each represent typical aspects of the technology, both good and bad. Evaluation of each prototype was done by investigating the architectural models themselves, from the point of view of a developer, by their overall performance, popularity in the Finnish job market, and implementation costs.

The research confirmed that none of the selected technologies is an absolute choice. Instead, the outcome highlighted which use cases would best benefit or be marred by each technology. Monolithic was found to be mainly suitable for tiny applications with no prospect of future development while becoming troublesome if used for anything more complicated. Microservice proved to be an optimal choice regarding re-usability, although requiring a rather high development and maintenance workload. The AWS Cloud was the most flexible solution, requiring more understanding of the AWS itself than actual software development; however, the downside was possible high costs if not properly designed and optimized due to its pricing model and the finality of choosing a cloud provider.

Keywords/tags (subjects)

Software architecture, software development

Miscellaneous (Confidential information)

-

jamk

Description

Pikkumäki, Toni

Monoliittisen, Mikropalvelun, ja Pilvi sovelluksen kehityksen vertailu

Jyväskylä: JAMK Jyväskylän Ammattikorkeakoulu, Huhtikuu 2023, 106 sivua.

Ylemmän Ammattikorkean tutkinto Tekniikan alalla. Full Stack Software Development. YAMK Opinnäytetyö.

Verkkojulkaisulupa myönnetty: Kyllä

Julkaisun kieli: Englanti

Tiivistelmä

Moderni sovelluskehitys tarjoaa paljon vaihtoehtoja. Ei ole yhtä teknologiaa, joka olisi muita parempi. Sen sijaan, eri teknologioilla on omat etunsa ja haasteensa. Yksi keino sovelluskehitysteknologian valinnassa on määritellä ohjelmiston tärkein prosessi, ja suorittaa vertailu kohdeteknologioiden välillä.

Tähän tutkimukseen valittiin kolme teknologiaa: .NET-monoliittinen, Java Spring Boot -mikropalvelu ja AWS-pilvisovellus. Valinnat edustavat sekä eri ohjelmointikieliä että ohjelmistoarkkitehtuureja, ja vertailun painopiste oli jälkimmäisessä. Päätavoitteena oli löytää näiden valintojen sekä parhaat että huonot puolet, sekä vertailla teknologioita keskenään. Tutkimus keskittyi sovellusten toteutuslogiikkaan, ilman testaus- ja DevOps-osuuksia. Jokaisesta teknologiasta rakennettiin oma prototyyppinsä, minkä tavoitteena oli havainnollistaa näiden sekä edut että haitat. Teknologioiden arviointi tehtiin tarkastelemalla arkkitehtuureja itsessään, sovelluskehittäjän näkökulmasta, suoriutumisen pohjalta, suosiolla Suomen työnhaussa sekä kehityskustannuksista.

Tutkimus vahvisti väitettä, että yksikään teknologioista ei ole yksinään paras valinta. Lopputulokset toivat esiin tapauksia, jotka joko parhaiten tai huonoiten sopivat kyseisiin teknologioihin. Monoliittinen oli soveltuvin pieniin sovelluksiin, joita ei tulla jatkokehittämään, mutta huono yhtään monimutkaisemman kanssa. Mikropalvelu osoittautui uudelleenkäytettävimmäksi, vaikkakin työläimmäksi toteuttaa ja ylläpitää. AWS-pilvi oli joustavin vaihtoehto vaatien enemmänkin ymmärrystä itse AWS-rajapinnasta sovelluskehityksen sijaan, jonka haittana on se, että kalliin laskun välttäminen hinnoittelumallin vuoksi vaatii tarkkuutta kehittäessä, sekä pilvitarjoajan valinnan vaihdon vaikeus.

Avainsanat (asiasanat)

Ohjelmistoarkkitehtuuri, ohjelmistokehitys

Muut tiedot (salassa pidettävät liitteet)

-

Contents

Figures

Tables

Terms and abbreviations

IDE	Integrated Development Environment.
Container	A technology used to run applications in a self-contained environment.
CI/CD	Continuous Integration and Delivery.
UI	User Interface.
HTTP/HTTPS	Hypertext Transfer Protocol.
API	Application Programming Interface.
API Key	Authentication used with an API.
API Gateway	An API that acts as an entry point or a middle layer for other API calls.
Request	A call to an API.
Response	An outcome from an API request.
200 OK	A HTTP response status code indicating success.
REST	Representational State Transfer.
URL/URI	An address of a webpage or other resource.
IAM	Identity and Access management.
XML	Extensive Markup Language.
JSON	JavaScript Object Notation.
GitLab	A version control repository.
Factory/Provider	A class used to create or give access to other classes.
.NET	A software framework by Microsoft.
async Task	A .NET pattern to make an execution of a method asynchronous.
Spring Annotation	A @-prefixed class for configuration of another class, method, or field to configure its behaviour.
pom	XML file used by Maven to define a project.
Data Normalization	Collecting data of varying structures and unifying them into a shared format.
Postman	A software for sending of HTTP requests.

1 Introduction

This thesis concerns research and development with three distinct software architectural models: monolithic, microservice, and cloud. While these architectures are different, they can all still be used to develop the same software, although the final products from each will be quite dissimilar. The thesis details what these architectural models are, what the software developed with each look like, and how they stand up individually and when compared to each other.

The thesis is not commissioned by anyone; instead, it is created based on the past experiences of the thesis author for academic purposes only.

1.1 Thesis structure

The first chapter begins with an introduction and background of the thesis. This is followed by defining the goals and research questions, motivation and reasoning behind the selected subject, and methodology.

The second chapter forms the first half of the research. It is a literature review, collecting information relevant to the thesis subject from outside sources.

The third chapter details the development part of the thesis. Created prototypes are explained in detail, offering an overview of what and how they have been implemented.

The fourth chapter contains the second half of the research. Each developed prototype is evaluated, followed by an interpretation and discussion of evaluation results.

The fifth and final chapter summarizes the findings and draws conclusions based on the research done in previous chapters. The thesis ends by discussing the research and development, including suggestions for further research.

1.2 Original application

The core concept for the thesis is based on an actual application developed at work by the thesis author; however, the original application has no real role in the thesis beyond inspiring the topic.

Two separate versions of the original application have been implemented, the first in 2016 as a monolithic .NET WPF application and the second in 2021 as a pair of Java Spring Boot micro-services. The 2016 version was created to expedite data processing, whereas the 2021 version was implemented to automate it further.

The original application handles data integration from various third-party providers into the company's internal systems, an operation usually done once every month. Every third-party provider has a different data format and delivery method, whereas the data stored in the company's internal system is expected to be uniform. As such, the three main use cases of the application, as further illustrated by Figure 1, are:

1) Fetching data from a third-party system.
2) Normalization of the data into the shared format.
3) Sending the data into the company's system.

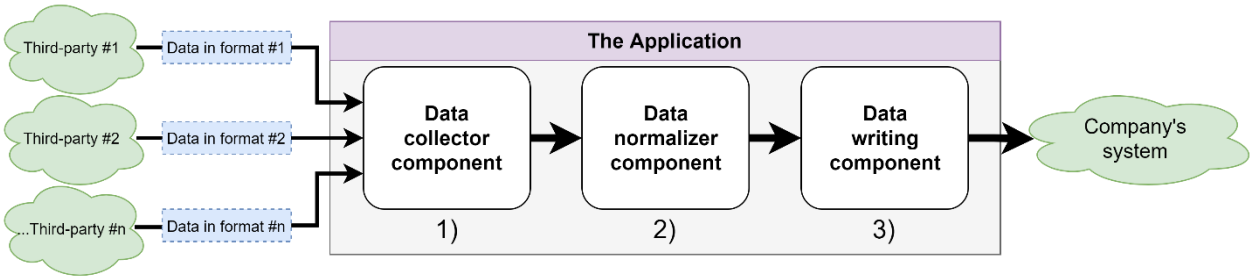


Figure 1. Data flow through the application

The need for the application arose in 2016 when an employee's manual handling of a single third-party provider's data was becoming unreasonably laborious. Further investigation into this problem revealed that most manual steps could be replaced by automation, leaving only a few key steps for an employee to perform. Although the application was initially designed to resolve data from a single third-party provider, it was soon expanded to support additional providers.

While the application implemented in 2016 simplified data processing, it still required an end user to operate it. Adding further automation would have required rebuilding the whole application from the ground up. Thus, it was decided in 2021 that implementing a new application with different technologies would be a better solution.

Although the application remake in 2021 was done with Spring Boot, it would have been possible with alternative technologies as well, leading to the topic of this thesis to explore some of these alternatives from a developer's perspective.

1.3 Purpose and objectives of the thesis

The main purpose of this thesis is to build and evaluate different software development technologies, with each technology representing a different software architecture model.

1.3.1 Objectives of thesis and research questions

Technologies selected for the thesis are .NET monolithic, Java Spring Boot microservice, and AWS Lambda with Python. These choices offer both different architectural models and development languages, although the focus will be on the former, with the latter providing only more variety and similarity to the original case from work.

The first objective is to build a prototype using each selected technology, implementing the core use case of the original application. The second objective is to evaluate built prototypes and the technologies used to implement them.

When reporting created prototypes, the objective is to offer an overview of their implementation. As a basic knowledge of selected technologies is expected from the reader, the purpose of the thesis is not to explain how selected technologies work, only how the prototypes can be implemented with them.

This thesis aims to answer two main research questions "*What are the strengths and weaknesses of each selected technology?*" and "*How do the selected technologies compare with each other?*".

1.3.2 Reasoning for research and development and expected results

When selecting between technologies and software architectures, modern software development offers a wide variety of choices. No single technology surpasses all others; instead, all technologies have their strengths and weaknesses. One good way to decide which one to choose is to determine the primary use cases of the application and compare how each technology candidate performs.

A more in-depth comparison takes time and resources; as such, many developers will choose to refer to pre-existing materials instead. By comparing technologies with a fairly common use case, the results of this thesis can offer help to those struggling with related technology decisions, making software developers and architects, as well as project managers, the main target audience of this thesis.

While the original concept of this thesis originates from an actual working-life project, the purpose is not to determine whether that specific selection was the right choice. The thesis will analyze technologies from a shared and neutral starting position, whereas the selection made at work was influenced by technologies already used within the company.

The personal interests of the thesis author contributed to the selection of this topic as well. Remaking an application at work made the author curious to find out how the application would have turned out if implemented either with the original technology or with an even more modern approach. Additionally, by investigating and implementing prototypes with the three selected technologies, the author will gain increased professional expertise.

The outcome of the thesis is expected to contain an introduction to select technologies, an evaluation of their overall capabilities, and a comparison between them.

1.3.3 Reasoning for technology choices

To offer more variety and match original applications, each technology choice contains different architectural models and programming languages.

.NET monolithic

.NET monolithic with C# was selected to match the original application from 2016. The goal is not to focus on the past implementation; instead, the objective is to explore how the more automated version from 2021 could have been built using the original technology.

Regardless of the growing popularity of non-monolithic solutions, they are not always the best solution; due to their simplicity, monolithic applications are still developed by various organizations,

and there are even cases where an application has been migrated from non-monolithic to monolithic, meaning that monolithic solutions are still relevant (Mendonça, Box, Manolache, & Ryan, 2021; Microsoft, 2021).

Java Spring Boot microservice

Java Spring Boot microservice was selected to match the application implemented in 2021. Due to its scalability and increase in popularity, microservice architecture has become relevant in modern software development, making it an optimal target for evaluation.

AWS Lambda with Python

Because DevOps and agile development models are undergoing growth in popularity, serverless cloud solution provides a fresh alternative to the other two technology choices (Wadhwani & Loomba, 2021).

Based on Datadog (2022) survey, AWS Lambda is the leading serverless provider, with Python as the most used programming language, although NodeJS is only marginally less popular. Due to their proven popularity, AWS Lambda with Python was selected for evaluation.

1.4 Methodology

1.4.1 Data collection method

Software prototyping is the practice of building a preliminary or simplified version of an application, which can be anything from a simple representation of a planned product to complex software with actual functionality. This method is often used to demonstrate different aspects of an application for a specific purpose, for instance, to gather feedback about a planned feature or sell a product that does not yet exist. Software prototyping can also be used as a form of software development, with the prototype eventually becoming the full published product.

Quilliam (2022) explains that software prototyping can be approached with various methods, the four main ones being: incremental, extreme, evolutionary, and throwaway prototyping. Incremental prototyping allows fast development by creating separate instances in multiple development

cycles of the prototype, eventually combining them into one final product. Extreme prototyping method is mainly meant for the development of a website, building the site in three stages, with the first stage creating web pages, the second stage adding functionality, and the final stage adding the service layer. Evolutionary prototyping begins with the creation of a core prototype, which is then extended and refined with each future iteration, making this method especially useful if requirements for the software are not fully known at the start of the development. Throwaway prototyping differs from other mentioned methods by abandoning the prototype after testing, whereas other methods often continue building the prototype into the final product. Because there is no need to refine the prototype past the initial implementation, throwaway prototyping is a good choice for any proof-of-concept creation.

While the prototyping process is often described as first building the prototype, then collecting feedback from end-users, it is not restricted to this; the end-user feedback can be replaced by the evaluation segment of the thesis.

Applications the thesis is based on have dependencies and connections to other services and components, which is only ideal for comparing technologies if these dependencies are also part of the comparison. Since assets from original applications cannot be used within the thesis, using prototypes with shared requirements, and focusing on the main-use case only will offer a more equal basis for evaluation. Hence, prototyping is the best method of choice for this thesis, and since there will be no further iterations of prototypes, the throwaway model is the one to use.

1.4.2 Evaluation method

Evaluation is done once all prototypes are complete, and since the intent of the thesis is not to build new software, their functionality is only one part for evaluation; additional areas to evaluate are, for instance, used development methods and the technologies themselves. This can be categorized as “Outcome evaluation”, which is a summative type of evaluation used to determine to what extent the project’s goals have been accomplished (Evaluation: What is it and why do it?, n.d.).

The ultimate goal of analyzing built prototypes is to provide an overall understanding of each technology's strengths and weaknesses in a way that allows for independent evaluation of each prototype, as well as comparative evaluation between prototypes, which is done with a mix of qualitative and quantitative methods. For instance, a quantitative method is used to evaluate data collected from prototypes, and a qualitative method is used to assess each prototype from a developer's point of view.

1.5 Ethicality

All software developers are allowed to have their personal preferences when it comes to development languages and models. However, understanding one's own inclinations helps to keep them from affecting the evaluation process.

The thesis author has varying levels of experience with selected technologies. To avoid this significantly impacting prototype quality, detailed specifications on how prototypes must be built and when they can be considered ready for evaluation are specified.

There is always a margin of error when analyzing collected data. While there is no calculation on what the actual margin of error is, the possibility for such is considered when discussing evaluation and its outcome.

2 Literature Review

An investigation of technologies chosen for prototyping: Monolithic, Microservice, and Cloud. Before moving into prototype implementations, it should be understood what these technologies are; thus, the main goal of this review is to define each of them. Properly characterizing each technology will assist when answering the main research questions, as their definitions already highlight some of their strengths and weaknesses, which can be used as focus points when evaluating prototypes.

2.1 Monolithic application

An application containing all logic within itself is called monolithic, usually consisting of a single code base and deployed as an isolated unit, often executed within a singular process or container (Microsoft, 2021; Kanjilal, 2020). However, being monolithic does not mean that an application cannot be internally built from different components, for instance, having separate layers for user interface and business logic. Incidentally, not all of these components have to be part of the application proper; like any other application model, monolithic applications can still make use of external libraries.

When a solution is either small enough or not expected to receive significant updates after initial implementation, a monolithic approach can be beneficial; after all, designing and developing a monolithic application is faster and easier than some of the more complicated solutions (Kanjilal, 2020). Handling application-wide information, such as configurations and authentication, becomes simpler when relevant data is readily available for all components within the shared codebase. Due to the solution executing as a single entity, its components typically have a shared memory; thus, there is no need to load or fetch shared data in multiple places, leading to a better overall performance in this regard, unlike solutions where each component would execute separately.

These factors have made the monolithic development model popular in various organizations; however, while successful for some, others have been faced with their limitations (Microsoft, 2021). For already extensive solutions or small applications that begin to grow, a monolithic architecture will begin to show its challenges.

When all components of an application are more or less dependent on each other, the application as a whole becomes more complicated with each revision, making development and deployment more expensive and time-consuming and leading to more error-prone solutions (Microsoft, 2021; Kanjilal, 2020). Whenever even a tiny part of the application is changed, the whole solution must be re-tested and re-deployed; additionally, depending on the used deployment process, this may result in downtime for the entire application.

2.1.1 Modular Monolith

An architectural model called Modular Monolith exists as a hybrid between modular and microservice, separating logic into distinct modules that communicate via APIs, with each module representing a specific feature (Agamez, n.d.; Maayan, 2022). A module has its individual business logic and, if necessary, its own UI and data access layers. Modules have minimal dependencies between each other, although they may contain a shared database. The whole application, with all of its modules, can be published as a single entity instead of publishing and maintaining each module separately.

With no need for each module to be fully independent, it trades the greater flexibility of the microservice for better performance while gaining the more straightforward design and implementation patterns of the monolithic (Agamez, n.d.; Maayan, 2022). Ultimately, the modular monolith aims to take the best parts from the monolithic and microservice architecture while losing some of their benefits in the process.

2.2 Microservice

While the core concept of microservice has existed for far longer, the term was first introduced in 2005 by Dr. Peter Rodgers as “Micro-Web-Services” during his presentation at a conference on cloud computing (Foote, 2021). The more streamlined term “Microservice” was introduced in 2011 by participants of a workshop for software architects before officially embracing the name a year later, as they felt it best characterized that architectural style.

2.2.1 Defining a microservice

Microservice is not a technology that has been designed, defined, or published by any single organization or other groups of people; instead, the concept of microservice has gradually evolved over time. As a result, defining what a microservice is has become more complex; after all, many have already defined microservice in their own ways.

Many of the existing definitions contain shared elements and concepts, some emphasizing specific aspects over others. As such, microservice can be defined as a sum of other definitions, forming a

more complex picture when put together. This is done by collecting and combining key aspects from existing microservice definitions into a unified summary.

Componentization

The base idea of a microservice is to migrate from an earlier way of thinking of software as a single entity into an architecture where logic is split between small and independent services (Fowler, 2014). When there is a need to adjust a specific component, only its respective service needs to be updated instead of restarting and redeploying a whole application (Ashtari, 2022).

As such, each service can be viewed as its own small application, making microservices by design very flexible because they can be developed, deployed, maintained, and even replaced independently (Fowler, 2014). However, this does not mean that services cannot be aware of and make use of each other.

Failure tolerant

Microservice application is by nature resistant to failures, provided that the application and services have been built with such a nature in mind (Fowler, 2014). While a single service becomes un-available, it does not mean that all services in the platform have failed. By building client and service-to-service connections to anticipate possible failures, the overall effect of the failure can be minimized or even circumvented via rerouting traffic to other services (Ashtari, 2022; Fowler, 2014).

This emphasizes the importance of automatic service restoration and robust service management systems; bringing the service back to running as fast as possible is vital, and the more critical the service is, the more noticeable its downtime will be (Fowler, 2014). In a well-formed service platform, tracking the root of the error is simplified since the application is already composed of separately running entities (MW Team, 2023).

Business capabilities

Development teams are often organized by technology or developer skillset, for instance, with different teams for UI and server-side, which can lead to complications when development requires

actions from multiple teams (Fowler, 2014; IBM Cloud Education, 2021). With microservice architecture, although not restricted to it, the preferred way of organization is by business capability. One team, consisting of people with multiple skill sets, is in charge of a single product consisting of multiple interconnected services, leading to more agile development.

Microservice should not be approached as a typical project, where a development team creates a product, then moves on to the next project, leaving the maintenance up to others (Fowler, 2014). Instead, a microservice should be treated as a product, with the original development team as the owner and taking complete responsibility for the entirety of its lifetime.

Having various teams working on different services at the same time, instead of all working on a single application, requires less coordination and is less error-prone, leading to faster and more efficient development cycles (MW Team, 2023).

Continuous integration and delivery

Microservice is expected to make use of various available continuous integration and delivery (CI/CD) solutions; otherwise, developing, publishing, and maintaining a large cluster of microservices would become too cumbersome in the long run. (IBM Cloud Education, 2021). With various vendors offering easy-to-use cloud-based solutions for CI/CD, this does not mean that microservices cannot be deployed on more traditional servers and without effective CI/CD; however, cloud-based and automated deployment does coincide more with the initial concept.

Communication via API

Communication between microservices, or from other software to microservices, is usually done via API; most commonly, a REST API that is accessed via HTTP(S), either directly or via API gateways (Fowler, 2014; IBM Cloud Education, 2021). From the API's point of view, a smartly designed microservice has a specific purpose, not offering anything that is not relevant to its purpose. This does not mean that the internal functionalities of the microservice have to be as simple; while the API should only expose intended functionalities, the service itself can be far more complex.

However, having a single microservice perform too many operations, be they internal functionalities or those exposed by the API, would begin to break the initial purpose of the microservice. Lacking any unified guidelines on the scope of a single microservice, it falls into a gray area to define when a microservice has become too complex to truly be considered one anymore.

Independent technology stack

Because APIs used to communicate with a microservice are technology-agnostic, this allows a microservice itself to exist as technology-agnostic (IBM Cloud Education, 2021). Because a single service is self-contained, it can be developed using any programming language and contain any data model possible. The choice of a microservice's internal technology should be irrelevant from the point of view of any other application; after all, other applications are not, nor should be, aware of anything that happens below the API layer.

Developers do not have to be restricted by existing technologies; when considering which to select, a choice can be made based on what works best with the microservice in question and its intended functionalities (MW Team, 2023). This additionally allows more flexibility for a company's employees by not restricting all programmers to a few select technologies.

2.2.2 Microservice architecture

Martin Fowler (2014) demonstrates that an effective way to explain microservice architectural style is by comparing it to a monolithic style. Commonly, central parts of a web-based application are client, server-side, and database. In a monolithic application, these would be bundled into a single process containing all functionalities required to execute the whole application. A microservice style changes this by further separating the client-side from the backend and dividing the backend itself, placing each distinct feature into a separate service.

A common way of communication via microservice API layers is with HTTP Request and Response (Richardson, n.d.). A client sends a request to a server, which then executes its internal logic, eventually returning a Response. Whereas in a monolithic solution, the server receiving requests should be the same, this may not be the case in a well-designed microservice solution. The design

philosophy of the client-side is affected as well; for instance, a properly designed client should mirror the backend's separation into distinct components. Because of this separation, the pattern of Request and Response becomes asynchronous; with a client sending multiple requests to different services, it is never certain that responses are returned in the same order.

As a more concrete example, an HTTP page authenticates with one request while retrieving data with another. On a monolithic application, because both requests are handled by the same service, authentication information is readily available for the data retrieval operation; it would also be possible to combine both actions into a single request. In a microservice application, authentication and data retrieval would exist inside their individual services, with the latter service making its own request to the former one in order to verify the validity of given credentials. The benefit of fully separated services is, for instance, that the authentication service is not aware of what the authentication credentials are used for, making it easier for other services and clients to make use of the shared service. If there is ever a change in one service, the other does not have to be aware of this and can keep functioning as-is.

2.3 Cloud

Cloud architecture allows combining technologies and components to form a virtualized resource pool shared via a network, with scaling and optimization for actually needed capacities (Red Hat, 2022; VMware, n.d.).

A cloud can be used for many purposes, for instance, storing files, hosting servers, and running applications. One of the most popular ways for developing and hosting a software in cloud is called a serverless architecture, a model which, according to survey by Datadog (2022), over half of organization that operate in cloud are using.

2.3.1 Serverless architecture

Unlike its name suggests, serverless architecture has servers; instead, it allows developers to migrate responsibilities that come with hosting a server to a third-party provider (Datadog, 2021). Without the need to perform hardware and software upkeep for the server itself, developers have more time to focus on the actual development, increasing productivity and lessening the time it

takes to get new software to production. Due to the by-usage invoicing model, serverless architecture is, by nature, cost-efficient. Resources allocated to running serverless services can be automatically scaled based on current usage volumes.

A common pattern within the serverless architecture is to split applications into small individual functions; thus, the term function-as-a-service (FaaS) is often used to describe the serverless architecture, although FaaS is more of a model within the serverless architecture instead of being one and the same (Chandrakant, 2022; King, 2022). A FaaS architecture consists of using custom functions coupled with existing services from a cloud platform, with the execution of the application done via triggers and events.

A FaaS function is not unlike a microservice, as both prioritize independent modules over one huge application (King, 2022). FaaS servers can automatically start and shut down based on whether a function is currently executing, allowing server resources to be used only by functions that presently need them. This forms the common pricing model for FaaS, where the customer organization is billed by used resources only. As such, having FaaS functions as small and focused as possible will reduce the overall costs, in addition to making the code easier to develop and maintain.

As an example of a FaaS application process, a common starting point is the API layer, which is usually handled via an existing API Gateway service. The API Gateway then calls a custom function, which in turn triggers the execution of other functions and services. These will eventually lead to a desired outcome, be it a response from the API layer or some other final state.

Serverless architecture is not without downsides. Only the third-party provider can fix possible failures within the server infrastructure, and since there may be software from multiple customers running on the same server, application data could be exposed if the server is not configured well enough (Datadog, 2021).

Longer startup time for inactive services, restrictions on allowed languages or software versions, and difficulty in iterating integration tests can negatively affect the performance, development, and testing of a serverless application (Datadog, 2021; Chandrakant, 2022).

A serverless application essentially becomes locked to the cloud provider of choice (Datadog, 2021; King, 2022). Services that a custom application depends on, like databases and APIs, will not integrate well with their competitor's systems, and decoupling from any cloud provider altogether will require excessive work.

2.3.2 Other cloud architectures

Unlike FaaS, the infrastructure-as-a-service (IaaS) architecture leaves everything except the hosting of the physical server for developers (King, 2022). With IaaS, developers will have the deciding factor on all aspects of hosting the application, including the option to transfer some of these responsibilities to existing solutions like load balancers and other software.

More similar to FaaS is a platform-as-a-service (PaaS) architecture with the main difference that a PaaS application is published to the cloud as a whole, while a FaaS treats each function as an individual service (Chandrakant, 2022; King, 2022). PaaS architecture leaves both physical and operative server responsibilities for the cloud provider while giving developers control over the actual application hosting.

A PaaS requires fewer infrastructural configurations than IaaS, and in turn, FaaS requires fewer operative configurations than PaaS (Chandrakant, 2022; King, 2022). More responsibilities for developers also means they have better governance over their platform; thus, IaaS and PaaS can be more suitable than FaaS for constantly running or large applications (Datadog, 2021).

2.3.3 Cloud providers

The most popular cloud providers on February 2023 are Amazon Web Services (AWS) with 34% market share, Azure with 21%, and Google Cloud Platform (GCP) with 11% (Law, 2023). While there are a plethora of other providers, like Alibaba and IBM, with marked shares of 5% and 3%, respectively, their popularity has not yet reached the same levels as the top three.

AWS is a part of the larger Amazon corporation and the oldest of the most popular providers, originally introduced in 2004 and properly launched in 2006 (Pletcher, 2022). The platform originated

when Amazon shifted its internal development for reusable modules in 2000 and later conceived to monetize this concept.

Microsoft Azure was originally released in 2010 as Windows Azure; however, the first iteration was not satisfactory, with a more robust iteration released in 2013 (Pletcher, 2022). Microsoft began to prioritize the cloud and renamed the product to Microsoft Azure in 2014.

GCP began as a PaaS that was published in 2008 and eventually evolved into a cloud platform with FaaS features equaling its competitors while still offering the PaaS features as a product called an App Engine (Pletcher, 2022; Chandrakant, 2022).

3 Prototypes

Three prototypes are built to resolve the core use case of the original application: the synchronization and normalization of data. This section details the specifications and requirements the prototypes must meet and provides an overview of each implementation.

Code files most relevant to the core functionality of each prototype implementation are included in the Appendices section. Any non-relevant feature, like logging, is omitted from these examples.

3.1 Prototype technical specifications

To accurately compare different prototypes, they must be comparable to begin with. This can be asserted by finding key aspects of the application and defining strict requirements that every prototype must meet. As there are differences between each technology, additional per-technology requirements will define what is expected of each prototype at both architectural and functional levels.

Requirements used when building prototypes are not part of the actual evaluation; these merely exist to assert the quality of prototypes for further evaluation. The outcome of the evaluation cannot be considered reliable if the prototypes being analyzed are not of equal overall quality.

3.1.1 Development patterns

The purpose of prototypes is to offer an accurate overview of their chosen technologies, both good and bad aspects. Each technology and software architecture model will have its own shortcomings, most of which could be circumvented or mitigated with clever use of code or pre-existing additional components. As implementations should present the core architecture as well as possible, they should not use these workarounds.

Prototypes must follow similar coding patterns during implementation. For instance, reading and handling data could be done with different approaches, as shown in Figure 2, where the first approach utilizes simpler and easier-to-understand code, in contrast to the second approach, which offers better memory optimization. While the example is written in C#, the similar design philosophy still applies to other languages as well, and for evaluation to be more accurate, it does not matter which of the approaches is used as long as all prototypes follow the same pattern; otherwise, one prototype would have an advantage over the others at the code level.

```
// Example 1:
//  - Reading the whole file at once.
//  - Handling lines after they have been read.
List<string> lines = File.ReadLines(FILE).ToList();
lines.ForEach(HandleLine);

// Example 2:
//  - Reading the file one line at a time.
//  - Handling each line before moving on to the next one.
using (FileStream fileStream
    = new FileStream(FILE, FileMode.Open, FileAccess.Read))
using (StreamReader reader = new StreamReader(fileStream))
{
    while (!reader.EndOfStream)
    {
        string line = reader.ReadLine();
        HandleLine(line);
    }
}
```

Figure 2. File reading examples with C#

3.1.2 Shared requirements

All prototypes must implement the following features:

- The primary execution process of the prototype must be as follows:
  - Data is fetched from a third-party source.
  - Data is normalized.
  - Data is committed to the output system, one line at a time.
- There must be a single-entry point to call, which will initiate the whole execution process of the application. The entry point should take both the third-party source and the target data to import as an argument.
- Prototype must have its internal configuration for the third party, making the data fetching possible without having to provide any additional third-party-related information.
- Adding support for a new third-party source must be relatively easy, requiring active development only for the data collector and normalizer features.
- Due to simulating a backend integration process, prototype is not required to have any graphical user interface.
- Prototype must use the latest stable version available for each selected technology at the moment of implementation.

The third-party source to test prototypes with is JSON files located at JAMK GitLab of the thesis Author. While the source could be anything from a file in the local system to a REST service, GitLab was chosen for its free availability for JAMK students and simple configurations. The output system is a single REST endpoint that validates that the sent data is of the expected format.

To offer more variety for different prototype components, both format and expected delivery methods are different between source and output systems (Figure 3 and Figure 4). While data in the GitLab source is contained as a single file, the output system will expect the data to be delivered one line at a time.

```
GitLab Source Format:
[
  {
    "ID": "string",
    "Category": "string",
    "Type": "string",
    "Value": "number",
    "ValueOn": "EPOCH date and time as integer"
  }
]

Expected Output Format:
{
  "Name" : "string",
  "Value" : "number",
  "Timestamp" : "ISO8601 date and time as string"
}
```

Figure 3. GitLab source and expected output formats

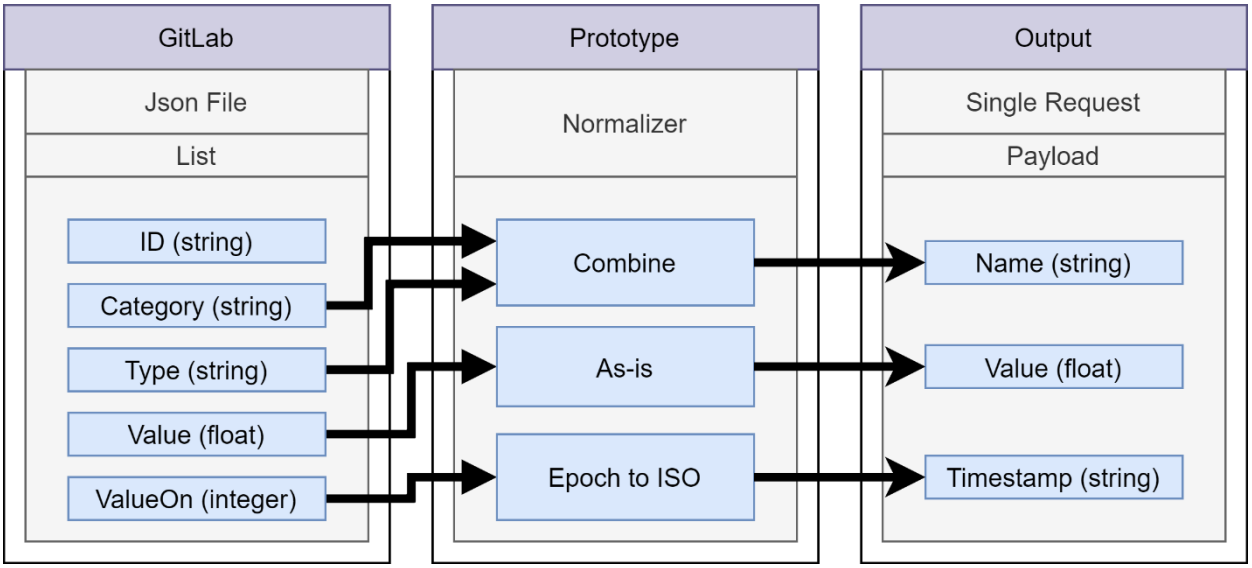


Figure 4. Data flow from Source to Output

3.1.3 Monolithic prototype requirements

The main objective of the monolithic prototype is to showcase a self-contained application implemented as a single solution.

The monolithic prototype must meet the following requirements:

- Prototype must be implemented with C# and .NET core.
- Application must be monolithic, having only a single executing process and shared memory.
- Entry point of the application must be a call from Windows Command Line or Windows Task Scheduler.

3.1.4 Microservice prototype requirements

The microservice prototype focuses on building a solution representing a typical microservice platform with individual services developed and functioning independently.

The microservice prototype must meet the following requirements:

- Prototype must be implemented with Java Spring Boot.
- Data-collector and data-sending must be separate microservices with REST endpoints for communication.
- Data-collector service has to be unaware of what the data is used for.
- Data-sending service must be unaware of the source data formats and call the data-collector service for normalized data.
- Both services must use a service registry. The purpose of the prototype is not to evaluate the service registry; this requirement only exists to simulate how microservices would discover each other in a production environment.
- Entry point of the whole process must be an endpoint of the data-sending service.

3.1.5 Cloud prototype requirements

The purpose of the cloud prototype is not to build a complex Python application; instead, the focus is on making a workable solution that makes heavy use of the wide variety of already available AWS services, thus allowing the final product to be built with a lesser amount of own code.

Furthermore, the main platform for custom code execution in AWS is the Lambda service. Thus, all Python code parts of the Cloud Prototype must be designed to be executed via the Lambda service.

The cloud prototype must meet the following requirements:

- Prototype must be implemented with Python.
- Prototype must run on AWS Lambda.
- Prototype must make use of built-in AWS features and synergies between them.
- Entry point of the whole process must be AWS API Gateway.

3.2 Monolithic prototype implementation

The monolithic prototype is a .NET console application written with C#. The .NET framework has multiple implementations to choose from, with .NET Framework as the more robust choice and .NET Core as a more lightweight alternative (Microsoft, 2022b). The latest version of .NET Core (3.1) is chosen due to the prototype not requiring any advanced features of the .NET Framework.

3.2.1 Development environment

The Integrated Development Environment (IDE) chosen for the .NET prototype is Visual Studio 2022 Community Edition. This is the obvious choice due to .NET as a technology being officially supported by Microsoft, with Visual Studio as the recommended IDE and the Community Edition as the free version targeted for individual use (Microsoft, 2022a). Due to the extensive features of Visual Studio, no other software is required for implementation.

3.2.2 Prototype structure

A regular .NET application consists of a top-level Solution containing any number of interconnected projects. The .NET prototype is separated into three distinct projects, each representing a different application layer: Console Client, Business, and Model. The relation between these projects is visualized in Figure 5.

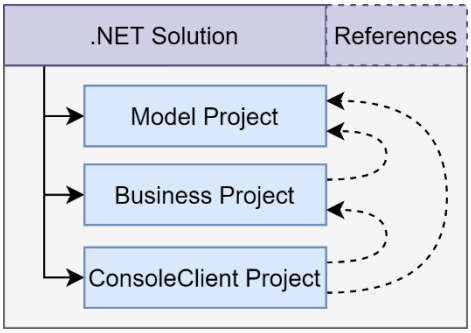


Figure 5. Monolithic prototype project structure and relations

Due to the monolithic nature, the whole application runs within a single process, with the core use-case illustrated in Figure 6. With only the single process, sharing common components becomes more simplified. For instance, all projects have access to the Model project; thus, keeping data objects constant throughout the data-handling process requires no additional work.

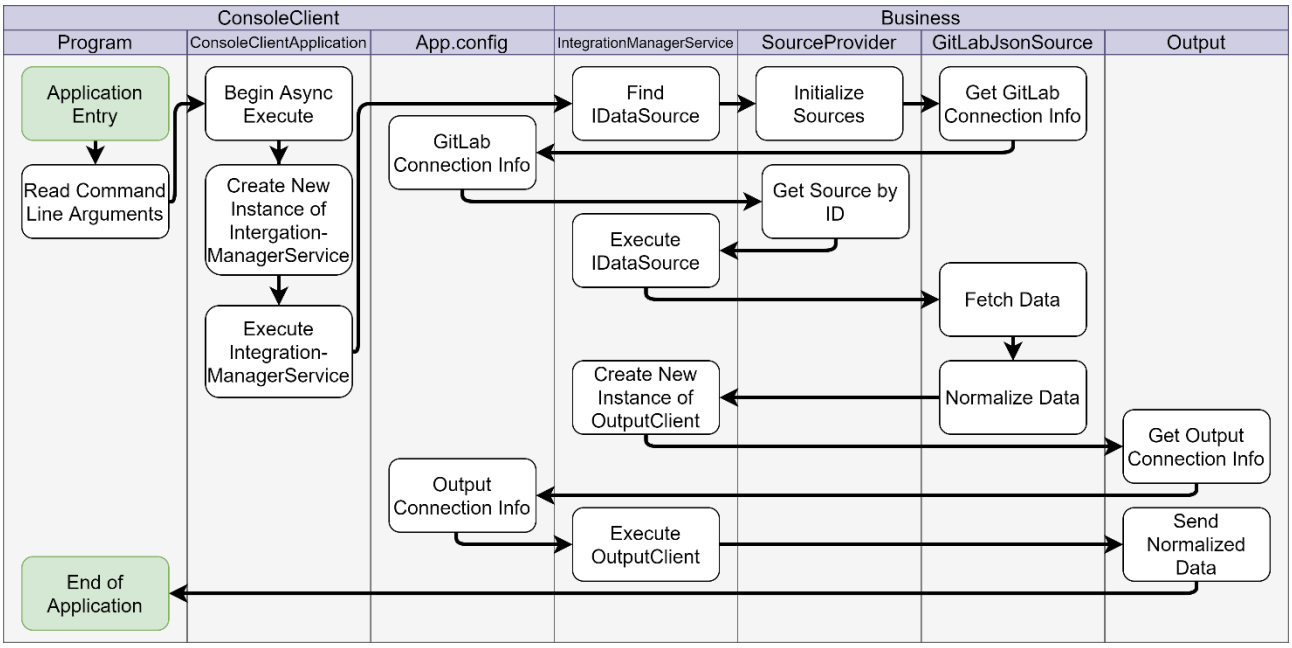


Figure 6. Monolithic prototype execution flow

3.2.3 Components overview

An overview of components that make up the monolithic prototype. The presentation is split to match projects and namespaces within the .NET solution, with the *business* project shown in the order the monolithic application invokes it.

The monolithic prototype uses a custom wrapper of the console application's inherent logging features; however, these are left out of the overview for not being relevant to the actual functionality of the application.

Application configuration file

Following the default .NET pattern, the configuration for the whole application is a file called *App.config*, which has been defined under the application entry project, and any component within the solution, be it business or client, has instant access to the configuration file via the built-in *ConfigurationManager* .NET class.

Each component has to know which settings they require while being allowed to remain oblivious of what other settings there may be. Implementation-wise, this simplifies the process, as there is no need to pass configurations from one component to another. The configuration file with the required settings for both GitLab and Output client access is shown in Figure 7.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
            <appSettings>
                        <!-- GitLab connection info. -->
                        <add key="GitLab_BaseUrl" value=""/>
                        <add key="GitLab_Project" value=""/>
                        <add key="GitLab_Folder"  value=""/>
                        <add key="GitLab_Branch"  value=""/>
                        <add key="GitLab_Token"   value=""/>

                        <!-- Output connection info. -->
                        <add key="Output_Url"   value=""/>
                        <add key="Output_Token" value=""/>
            </appSettings>
</configuration>
```

Figure 7. Monolithic prototype app.config file

Model project

The model project contains all data classes, interfaces, and enums. The purpose of this project is not to have any actual business logic; instead, it serves as the generic repository for entities to be used by other projects.

The most notable interface is the *IDataSource* (Figure 8), which is the sole representation of a data source that the majority of the application is aware of, except for classes that implement this interface.

```
/// <summary>A source for fetching of data from an external system.</summary>
public interface IDataSource
{
    /// <summary>Gets identifier that distinguishes this source.</summary>
    string ID { get; }

    /// <summary>Fetch and normalize data from this source.</summary>
    /// <param name="target">Determines which of the available data to fetch.</param>
    /// <returns>List of found data. Empty list if none is found.</returns>
    /// <exception cref="DataLoadingException">Thrown when loading fails.</exception>
    Task<List<NormalizedData>> FetchDataAsync(DataTarget target);
}
```

Figure 8. IDataSource interface of monolithic prototype

Console client project

The chosen main executable of the prototype is a console application. This allows flexibility on its intended usage and is the application entry type most easily ported to another.

The console application's primary purpose is to serve as the User Interface or interaction layer while leaving all actual logic for the rest of the application. The information on what data to fetch and from which source is given to the console as command line arguments, after which the console only needs to pass this information to the business layer and wait for the operation to end. To further distinguish between these purposes, the console client has been separated into two classes: *Program* and *ConsoleClientApplication*.

As shown in Figure 9, the Program is the application entry point, which receives the arguments, reads them, and passes them on to the *ConsoleClientApplication*, which invokes the business logic, as shown in Figure 10.

```
/// <summary>Execute the console application.</summary>
/// <param name="args">Arguments determining what to execute.</param>
static void ExecuteApplication(string[] args)
{
    string source;
    DataTarget dataTarget;
    ReadArgs(args, out source, out dataTarget); // Must have expected arguments.

    // Executing the application.
    ConsoleClientApplication application = new ConsoleClientApplication();
    application.Execute(source, dataTarget);
}
```

Figure 9. Key functionality of the monolithic prototype's Program class

```
/// <summary>
/// Execute the data integration process with the given source information.
/// </summary>
/// <param name="dataSourceID"><see cref="IDataSource.ID"/> to execute.</param>
/// <param name="dataTarget">Information on data to fetch.</param>
/// <returns>Async task.</returns>
private async Task ExecuteServiceAsync(string dataSourceID, DataTarget dataTarget)
{
    IntegrationManagerService service = new IntegrationManagerService();
    await service.ExecuteAsync(dataSourceID, dataTarget);
}
```

Figure 10. Key functionality of the monolithic prototype's ConsoleClientApplication class

As the first access point of the application, an additional purpose for the Console Client is error handling and event logging. In the case of a small .NET application like this prototype, the simplest solution would be to display logs in the console or to write them to the Windows Event Log.

Integration Manager Service of the Business Project

The core class of the Business project is the *IntegrationManagerService*, which is in charge of facilitating the Prototype's primary use case. As illustrated in Figure 11, the service begins by deciding which data source to use, loads the data, and finally triggers data sending to the output system.

```
/// <summary>Execute the integration process.</summary>
/// <param name="sourceID">
/// <see cref="IDataSource.ID"/> to integrate data from.
/// </param>
/// <param name="targetData">
/// Defines the target data at the <see cref="IDataSource"/> to integrate.
/// </param>
public async Task ExecuteAsync(string sourceID, DataTarget targetData)
{
    // Importing data from source system.
    IDataSource source = SourceProvider.Instance.Get(sourceID);
    List<NormalizedData> data = await source.FetchDataAsync(targetData);

    // Exporting data to output system.
    OutputClient outputAccess = new OutputClient(OutputConnectionInfo.FromConfig());
    await outputAccess.SendAsync(data);
}
```

Figure 11. Monolithic prototype's IntegrationManagerService execution

Source Provider of the Business Project

The *SourceProvider* class functions as both a factory and a provider for *IDataSource* implementing classes. Containing a singleton instance of itself, the provider self-initializes when first used, after which all future calls will use its already existing data sources.

When adding a new data source to the application, the only requirement is to register it to the provider, as the rest of the application will be aware only of the *SourceProvider* itself, not the individual sources it provides. The provider is used by calling its *Get* method with the identifier of the desired source, which is given to the application on startup via the console client, making the end user as the only other party aware of which source is used.

Having this class function as a provider instead of a factory is merely a design choice to prefer a single instance of every data source instead of creating a new one for each call. After all, in the scope of this prototype, this choice has no real effect on performance since the application is run separately for every execution.

GitLab JSON Source of the Business Project

GitLabJsonSource is the implementation of the *IDataSource*, in charge of fetching data from the GitLab and converting data into the normalized format. To simplify the JSON conversion process, the *GitLabJsonSource* contains its own DTO, the *GitlabJsonData*, which equals the data format available from the GitLab source.

The data fetching process forms an HTTP connection to GitLab and, using authentication details from the configuration file, downloads the JSON file, the name of which has been provided via the respective console argument (Figure 12). The process continues by reading the contents of the file and converting each row to the normalized format.

```
List<GitLabJsonData> result = null;
using (HttpClient client = new HttpClient())
using (HttpRequestMessage request =
    new HttpRequestMessage(HttpMethod.Get, this.GetUrl(target)))
{
    this.SetAuthorization(request); // Authorization header from config.
    using (HttpResponseMessage response = await client.SendAsync(request))
    {
        response.EnsureSuccessStatusCode();
        string responseBody = await response.Content.ReadAsStringAsync();
        result = JsonConvert.DeserializeObject<List<GitLabJsonData>>(responseBody);
    }
}
return result.Select(this.NormalizeData).ToList();
```

Figure 12. Monolithic prototype fetching data from GitLab

Output client of the Business Project

The final step in the application's execution process is the *OutputClient*, a self-contained class capable of both reading its own settings from the application's configuration file and sending data to the output system (Figure 13).

```
/// <summary>Send given data row to the output system.</summary>
/// <param name="data">Data to send.</param>
/// <param name="client">Open client connection to send with.</param>
/// <exception cref="DataSendingException">Thrown when sending fails.</exception>
private async Task SendAsync(NormalizedData data, HttpClient client)
{
    using (HttpRequestMessage request =
                new HttpRequestMessage(HttpMethod.Post, this.connectionInfo.Url))
    {
        this.SetAuthorization(request); // Authorization header from config.

        // The JsonContent with default settings will create a JSON
        // with output system's expected data format from the NormalizedData.
        request.Content = JsonContent.Create(data);

        using (HttpResponseMessage response = await client.SendAsync(request))
        {
            if (response.StatusCode != HttpStatusCode.OK)
            {
                string responseBody = await response.Content.ReadAsStringAsync();
                throw new DataSendingException(responseBody, response.StatusCode);
            }
        }
    }
}
```

Figure 13. Monolithic prototype sending data to the output system

Because the normalized data matches the format expected by the output system, the output client does not need to perform any additional data conversions. However, should the desired data format ever change, the self-contained nature of the output client means that only the client itself would require code changes, for instance, as a form of a simple data conversion function.

3.2.4 Running the prototype

Building and publishing the .NET project will result in an executable (.exe) file. The application can be executed manually, for instance, via the Windows Command Line or PowerShell, or the execution can also be automated, for example, via the built-in Task Scheduler feature of all modern Windows operating systems.

Figure 14 illustrates how the prototype can be executed for file01 of GitLab source, with the additional option of -s to skip confirmations and other user inputs.

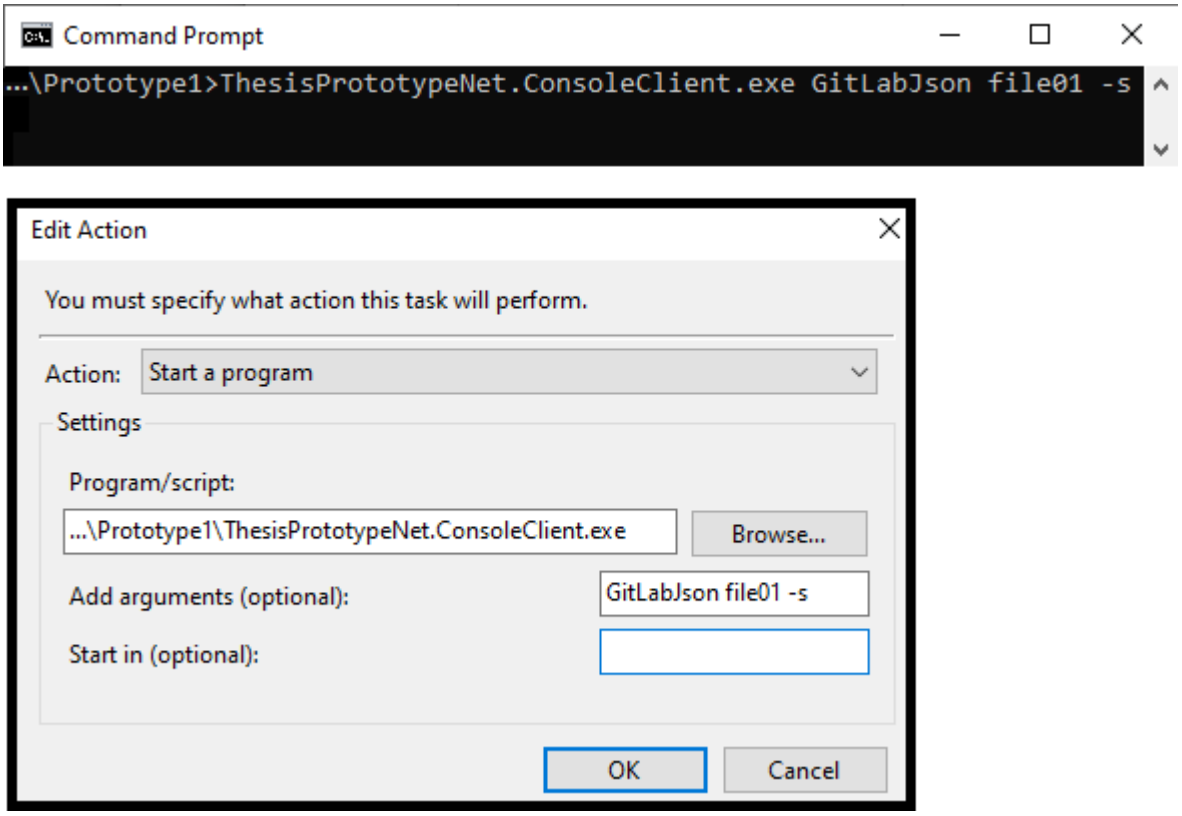


Figure 14. Execution of monolithic prototype via Command Prompt (upper) and Windows Task Scheduler (lower)

3.3 Microservice prototype Implementation

The microservice prototype is a Java Spring Boot application consisting of two separate micro-services. There is also a third service that acts as a service registry. All services use the latest version of Spring Boot and Java at the time of the implementation, that is spring-boot-starter-parent version 2.7.1 with Java 18.

Spring is a framework for simple dependency-injection-based Java development, while Spring Boot builds upon it, providing a robust framework for the development of REST-based applications, like microservices (InterviewBit, 2022).

Services of the prototype do not require authentication or authorization. This could be handled via a dedicated authentication service as part of the microservice platform; however, implementing such an extra service is outside the scope of this prototype.

3.3.1 Development environment

The Integrated Development Environment (IDE) chosen for the Spring Boot prototype is Visual Studio Code (VS Code). Due to its lightweight editor, extensive built-in features, and a multitude of available extension plugins, VS Code can be a good choice for almost any programming language (Microsoft, 2022c). While some of the more popular IDEs, like Eclipse, NetBeans, and IntelliJ have built-in support for Java features (vanshika4042, 2022) which the VS Code does not have, it makes up for this by providing respective features via extensions.

The tool chosen for software building and dependency management is Apache Maven, with its latest version for Windows at the time of the implementation being 3.8.6. The most popular software project management and build automation tools for Spring Boot are Maven and Gradle, both with their advantages and disadvantages. In the scope of this Prototype, differences between the two are minimal; thus, the selection between them falls to user preference.

3.3.2 Prototype structure

The prototype consists of three Spring Boot projects: the Data-Collector service, the Data-Handler service, and the service registry server.

As shown by Figure 15, data services follow the recommended Spring Boot layered project structure, where each layer exists as its own package (Ganeshchowdharysadanala, Spring Boot – Code Structure, 2021a):

- service
    - Application.java
- service.controllers
    - Controller1.java
    - Controller2.java
- service.services
    - Service1.java
    - Service2.java
- service.models
    - Model1.java
    - Model2.java
- service.[other layers]

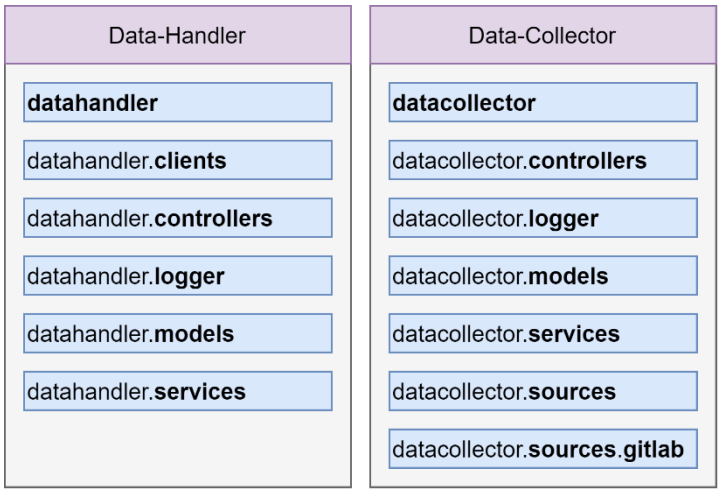


Figure 15. Microservice prototype Data-Handler and Data-Collector service layers

When these different services start up, they register themselves with the service registry, which is the Eureka server in this prototype. Afterward, all upcoming communication to and between the data services will be via HTTP requests, with the registry server providing the available URL. As further illustrated by Figure 16, the entry point for the prototype's main execution flow is the Data-Handler service, which uses the Data-Collector service for data fetching before sending data to the output system. Both data services are fully independent; no business logic or data transfer objects are shared between them.

Because the Data-Handler and Data-Collector services are intended to communicate via HTTP requests, they must include the Spring Web dependency, extending the default Spring Boot with REST features. Additionally, to register themselves and effectively communicate with each other via the discovery service, they must include its client dependency. The generated dependencies are the same for both Data Handler and Data Collector services, as further illustrated in Figure 17.

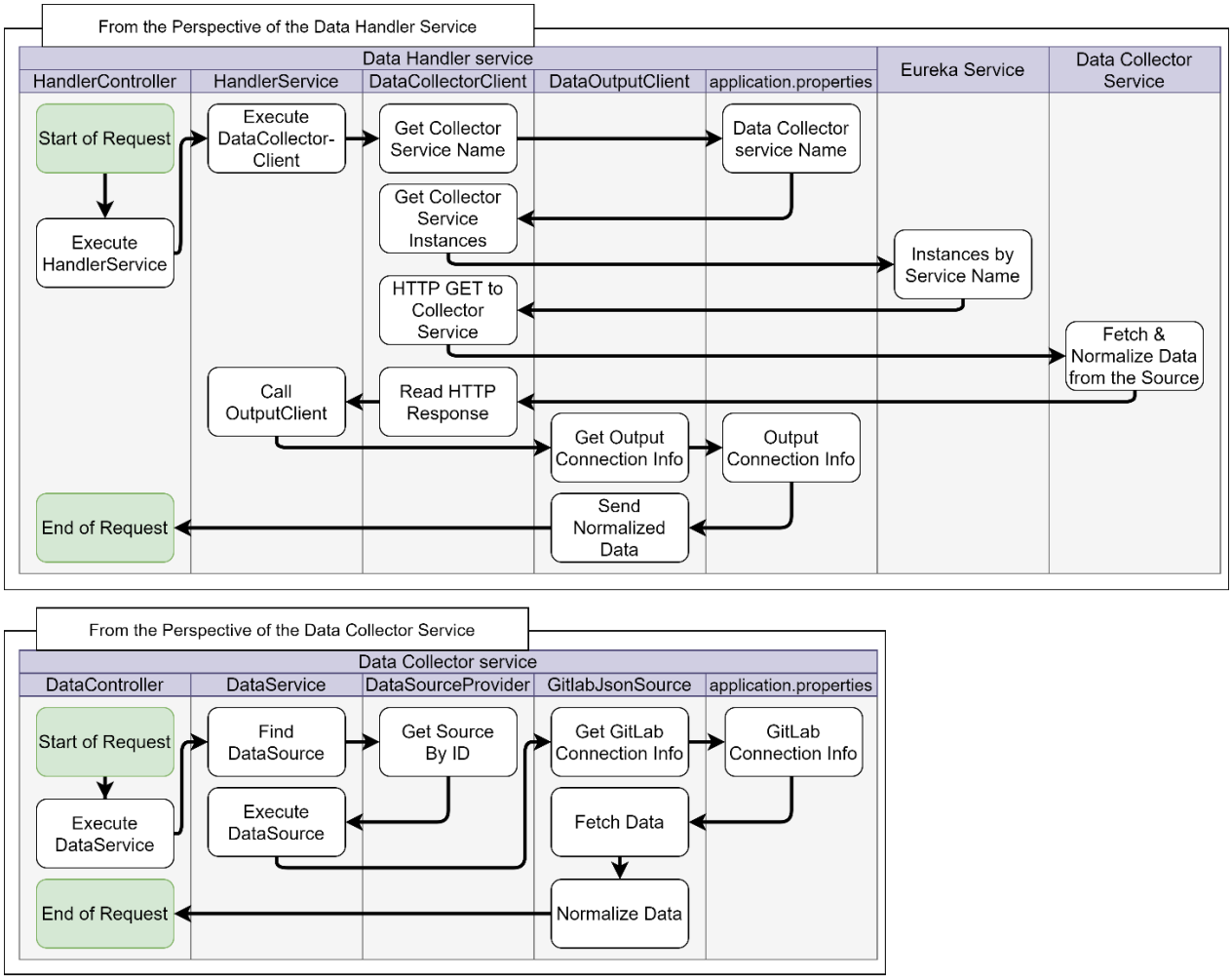


Figure 16. Microservice prototype's execution flow

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

Figure 17. Essential dependencies of microservice prototype's Data-Collector and Data-Handler services inside the pom.xml

Project structures with the latest version of Spring Boot and Java were created with a tool called Spring Initializr, the incorrect spelling of which is intentional and based on the similarly named HTML 5 tool (Ganeshchowdharysadanala, Spring Initializr, 2021b). The Initializr is available via a website, *https://start.spring.io*, as well as an extension for many IDEs. The user only has to fill in the basic information of the project and select which versions and alternatives of various offered components to use (Figure 18). Everything done by the Initializr could also be done manually; however, the officially supported tool makes starting a new project more straightforward and less error-prone, as a Spring Boot project requires more complex initial configurations than the other two prototypes not using a similar template builder.

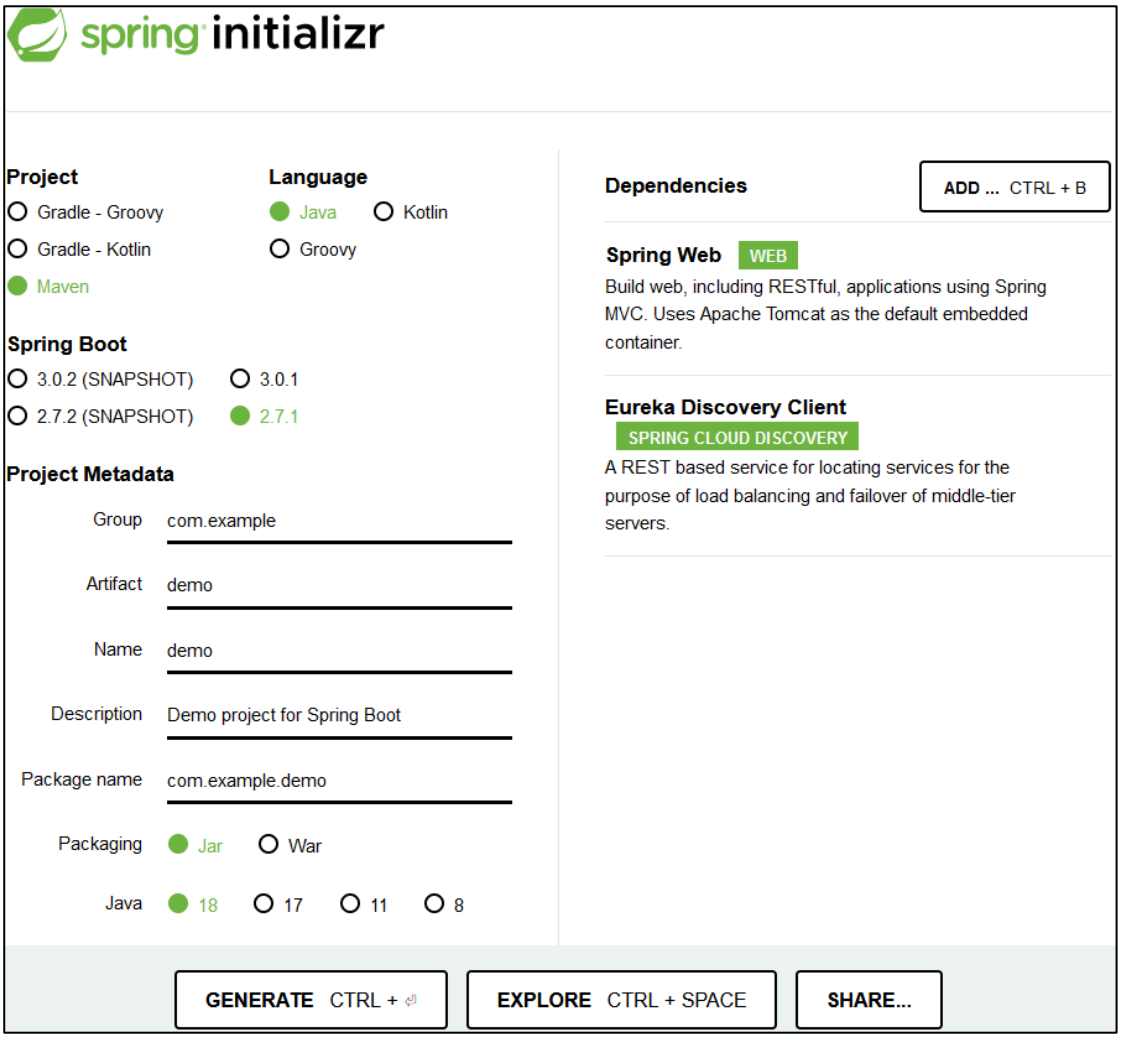


Figure 18. Spring Initializr web interface with selection equivalent to microservice prototype's services

3.3.3 Components overview

An overview of components that make up the microservice prototype. Due to its difference from other services, the Service Registry is explained under its own section. Presentation of Handler- and Collector-Services are done by service layer, with similar layers from both explained under the same chapter.

Behavior for the logger of Spring Boot is configured in application properties and used through the application via a custom wrapper. However, this is left out of the overview due to not being relevant to the main process.

Service registry

Microservice architecture aims to have a robust service-to-service communication that can easily withstand possible failures, which can be achieved via a service registry where all services register themselves to- and query for information on others (Long, 2015). Existing mechanisms for this are already provided by Spring Cloud, which contains support for some of the most popular service registries, like Eureka and Consul.

The Eureka server by Netflix is the service registry selected for the prototype. Because of the small nature of the prototype, the real benefit of this choice remains mostly unnoticed, and the selection of Eureka was made out of the author's preference due to past experience with it.

The Eureka server requires no custom logic to be coded, mainly working out of the box when generated via the Spring Initializr. As shown in Figure 19, the only line of code that separates the Eureka server from other Spring servers is the requirement to enable the eureka server functionality via the respective annotation. When running locally, the Eureka service is configured in the application.properties file (Figure 20) to run at a specific port with an optional pre-defined service name, with an additional setting to prevent the service from attempting to register itself in the registry (VMWare Tanzu, n.d.).

```
@SpringBootApplication
@EnableEurekaServer
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Figure 19. The whole Eureka service application of microservice prototype

```
server.port=8081
spring.application.name = eureka-server

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Figure 20. Microservice prototype Eureka service application.properties file

Application properties

The application.properties file is used to configure both port and root URL of services, as well as provide connection information required by the data source access of the Data-Collector service and output access of the Data-Handler service. Settings files for both services, with their required properties, are shown in Figure 21, using the localhost Eureka from Figure 20.

Because both Data-Handler and Data-Collector services have to register themselves to the service registry, both the registry server and the own name to register into it have to be included in application settings, although a default name would be used if not explicitly specified. Due to the simplistic nature of having Eureka as the service registry, the combination of this setting and the Eureka-Client dependency in *pom.xml* is all that is required for a service to function with Eureka. Additionally, the Data-Handler service must be aware of the Data-Collector service's name in the registry, which is stored in the Data-Hander's properties file.

```
Application.properties of Data-Collector service:
server.port=8082
server.servlet.context-path=/api/v1
eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka
spring.application.name=data-collector
aa8409.thesis.prototype.datacollector.gitlab.baseUrl=
aa8409.thesis.prototype.datacollector.gitlab.project=
aa8409.thesis.prototype.datacollector.gitlab.folder=
aa8409.thesis.prototype.datacollector.gitlab.branch=
aa8409.thesis.prototype.datacollector.gitlab.token=

Application.properties of Data-Handler service:
server.port=8083
server.servlet.context-path=/api/v1
eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka
spring.application.name=data-handler
aa8409.thesis.prototype.datahandler.collectorServiceName=data-collector
aa8409.thesis.prototype.datahandler.outputServiceUrl=
aa8409.thesis.prototype.datahandler.outputServiceToken=
```

Figure 21. Microservice prototype's Data-Collector and Data-Handler application.properties files

Models package

For both Data services, the *models* package contains the entity expected to be passed between the services and into the output system: *NormalizedData*. This is your basic Java class implementation of a Data Transfer Object (DTO). Due to the best-practice definition of a microservice expecting each service to function individually from the other, no shared package was used between these services, meaning a duplicate implementation of the same DTO is implemented in both services. However, this could have been circumvented by creating yet another Java project that both services use via Maven, a solution not used in the prototypes as it would have made both services dependent on the extra project.

The package also contains custom exceptions that may be thrown at various points of either service's execution. For instance, a custom *NotFoundException* is thrown by the Data-Service if attempting to fetch data from a non-existing source. The exception models will not be returned from a service; instead, they will only be used internally within each service for logging purposes and to decide which HTTP status code and message to return.

For Data-Collector, the *models* package also contains the *DataSource* interface, which is the representation of a data source as seen by most parts of the application, leaving only the interface's implementation in the *sources*-package aware of underlying differences between data sources (Figure 22).

```
/**
 * Data source for fetching of normalized data.
 */
public interface DataSource {

    /**
     * Gets unique identifier of this data source.
     * @return ID.
     */
    String getID();

    /**
     * Fetch data.
     *
     * @param targetName Defines the target data to fetch.
     * @return List of data. Throws exception if data fetching fails.
     */
    List<NormalizedData> fetchData(String target);
}
```

Figure 22. DataSource interface of microservice prototype

For Data-Handler, an *ExecutionTarget* DTO is included in the *models* package, which contains information on the desired data source and target, expected to be passed to the service when triggering its execution process, as will further be explained in the *controllers* package section.

Controllers package

Controllers define endpoints exposed by services and function as entry points of the application execution process. Spring Boot will automatically register the desired endpoint due to controller classes having the *@RestController* annotation and individual methods having either the *@GetMapping* or *@PostMapping* annotation. Controllers also have the *@RequestMapping*, which

will define how the URL from the service root to each endpoint within the controller will be formed. Other annotations are also available in Spring Boot but are not used by this prototype.

The controllers contain no proper business logic; apart from functioning as request entry and exit points, they are mainly in charge of passing received arguments to their respective *@Service* classes, the *DataService* and *HandlerService*.

For Data-Collector, the *DataController* is the entry point of the service, which is further detailed in Figure 23. Arguments are taken via query parameters, and since the method has no additional mapping, sending a GET request to the path defined on the class level via the *@RequestMapping*, will trigger the endpoint. The expected URL for the GET is https://service/api/v1/data?source_id=id&target_name=name.

```
 /** Controller for fetching normalized data. */
@RestController
@RequestMapping("/data")
public class DataController {

    /**
     * Fetch data from the given source.
     *
     * @param sourceID   Unique identifier of the data source to fetch the data
     *                   from.
     * @param targetName Defines the target data to fetch from the source.
     * @return List of data. Error status code if could not fetch data.
     */
    @GetMapping
    public ResponseEntity<List<NormalizedData>> get(
            @RequestParam(name = "source_id", required = true) String sourceID,
            @RequestParam(name = "target_name", required = true) String targetName)
    {
        List<NormalizedData> result = this.dataService.get(sourceID, targetName);
        return ResponseEntity.ok(result);
    }
}
```

Figure 23. DataController and its GET endpoint of the microservice prototype

For the Data-Handler service, the *HandlerController* and its *handle* method are the main entry point of the service, which is further detailed in Figure 24. This implementation differs from the similar endpoint of the Data-Collector service by receiving its arguments via the request body. In contrast, the collector received the equivalent information via request parameters without the additional DTO. This is done intentionally to showcase two different but equally valid approaches. Unlike the Data-Collector's controller, the *handle* method does not return any content on success, only the 200 HTTP status when everything went as expected. The expected URL for the POST is https://service/api/v1/handler.

```
/**
 * Controller for data handling execution triggering.
 */
@RestController
@RequestMapping("/handler")
public class HandlerController {

    /**
     * Handle data of given target.
     *
     * @param target Defines the data to handle.
     * @return OK when success.
     */
    @PostMapping
    public ResponseEntity<Void> handle(
            @RequestBody ExecutionTarget target)
    {
        this.handlerService.handle(target); // Throws an exception if fails.
        return ResponseEntity.ok().build();
    }
}
```

Figure 24. HandlerController and its POST endpoint of the microservice prototype

An additional class in the *controllers* package of both services is a *ControllerExceptionAdvice*, a spring *@ExceptionHandler* that converts custom exceptions from the *models* package into equivalent HTTP status codes and messages. This advice is automatically injected into all *@RestController* methods of the Spring service.

Services package

Classes annotated with the *@Service* serve as the core business logic layer, mainly in charge of facilitating the execution of deeper business logic layers.

This is the *DataService* for the Data-Collector service, a class in charge of determining which source to use via the *DataSourceProvider*, and then triggering the execution of the *DataSource* itself (Figure 25).

For the Data-Handler service, the *HandlerService* is the class in charge of triggering the data fetching from the Data-Collector service via the *DataCollectorClient* and sending that data into the output system via the *OutputClient* class (Figure 26).

```
/**
 * Fetch data from given source.
 *
 * @param sourceID   Unique identifier of the data source to fetch the data
 *                   from.
 * @param targetName Defines the target data to fetch from the source.
 * @return List of data. Throws exception if data fetcing fails.
 */
public List<NormalizedData> get(String sourceID, String targetName) {
    DataSource source = this.dataSourceProvider.get(sourceID);
    List<NormalizedData> result = source.fetchData(targetName);
    return result;
}
```

Figure 25. The core method of microservice prototype's DataService

```
/**
 * Handle data of the given target.
 *
 * @param target Defines the data to handle.
 * @return OK when success.
 */
public void handle(ExecutionTarget target) {
    List<NormalizedData> data = this.dataCollectorClient
            .getNormalizedData(target.getSourceID(), target.getTargetName());

    for (int index = 0; index < data.size(); index++) {
        this.outputClient.send(data.get(index));
    }
}
```

Figure 26. The core method of microservice prototype's HandlerService

Sources package of the Data-Collector service

The *sources* is a package specific to the Data-Collector service, containing logic for loading and normalization of data from external systems. Each source has its own sub-package, which in this prototype is *GitLab*. All source specific DTOs and logic are restricted to the sub-package, meaning that adding a new source only requires the creation of a new sub-package with its self-contained business logic.

DataSourceProvider is a Spring Boot component containing a list of all available *DataSource* implementations and a method to get them via their unique identifier. Due to the *@Autowired* functionality of Spring Boot, there is no need for explicit registration of sources to the provider; as long as a *DataSource* implementation is annotated with *@Component* or one of the other equivalents, the provider will receive all sources via dependency injection of the *DataSource* list, as shown in Figure 27.

```
/**
 * Create a new provider.
 *
 * @param sources All sources that can be provided.
 */
public DataSourceProvider(@Autowired List<DataSource> sources) {
    this.sources = sources;
}
```

Figure 27. DataSourceProvider of microservice prototype receiving sources via dependency injection

The *GitLabJsonSource*, in charge of fetching and normalizing data from GitLab, implements the *DataSource* interface and is located within a respective sub-package of the *sources* package. The data fetching is achieved using connection information from the application properties file, and the name of the file passed to the source as an argument (Figure 28). *GitLabJsonData* and *GitLab-ConnectionInfo* classes from the same package simplify this process by providing a data model equal to GitLab and logic for properties file reading.

```java
/**
 * Fetch data.
 *
 * @param targetName Defines the target data to fetch.
 * @return List of data. Throws exception if data fetcing fails.
 */
public List<NormalizedData> fetchData(String target) {
    // Preparing the connection information.
    GitLabConnectionInfo connectionInfo =
        GitLabConnectionInfo.fromEnvironment(this.environment);
    URI uri = this.getUri(connectionInfo, target);
    HttpHeaders headers = new HttpHeaders();
    headers.setBearerAuth(connectionInfo.getToken());
    HttpEntity<?> http = new HttpEntity<>(headers);

    List<GitLabJsonData> responseBody = null;
    try {
        // Fetching the data. Throws exception if fails.
        ResponseEntity<List<GitLabJsonData>> response = this.restTemplate
                .exchange(uri, HttpMethod.GET, http, TYPE_REFERENCE);
        responseBody = response.getBody();
    } catch (Exception exception) {
        throw new DataLoadingException(
                exception.getMessage(), this.getID(), target);
    }
    if (responseBody == null) {
        throw new DataLoadingException(
                "Response body is null.", this.getID(), target);
    }

    // Returning the normalized variant of the data.
    return responseBody.stream()
            .map(this::normalizeData)
            .collect(Collectors.toList());
}
```

Figure 28. Microservice prototype fetching data from GitLab

Clients package of the Data-Handler service

All logic for accessing other services from the Data-Handler service can be found in the *clients*-package.

The *DataCollectorClient* begins by fetching the actual endpoint location of the Data-Collector service from the service registry, which is done using the *DiscoveryClient*, a component available via the *springframework.cloud* package. This is done separately for each call to the client, thus allowing the discovery service to potentially give a different URL for different calls, if necessary (Figure 29). The client then performs a basic HTTP GET to the Data-Collector service's main endpoint, which returns the *NormalizedData*.

```
/**
 * Fetch normalized data from given source.
 *
 * @param sourceID   Unique identifier of the data source to fetch the data
 *                   from.
 * @param targetName Defines the target data to fetch from the source.
 * @return List of data. Error status code if could not fetch data.
 */
public List<NormalizedData> getNormalizedData(
            String sourceID, String targetName) {
    URI uri = getDataFetchUri(sourceID, targetName);
    HttpHeaders headers = new HttpHeaders();
    HttpEntity<?> http = new HttpEntity<>(headers);

    try {
        ResponseEntity<List<NormalizedData>> response = this.restTemplate
                .exchange(uri, HttpMethod.GET, http, TYPE_REFERENCE);
        return response.getBody();
    } catch (HttpStatusCodeException httpException) {
        throw new HttpException("Failed to fetch normalized data.",
            httpException.getStatusCode());
    }
}
```

Figure 29. Microservice prototype's Data-Handler service making a request to the Data-Collector service

The *DataOutputClient* is in charge of connecting to the output system. Fetching connection details from the *Application.properties*, the client performs basic HTTP POST requests to the output system with data to send provided as an argument to the client (Figure 30).

```
HttpHeaders headers = new HttpHeaders();
if(StringUtils.isNotBlank(token)){
    headers.setBearerAuth(token);
}
HttpEntity<NormalizedData> http =
            new HttpEntity<>(normalizedData, headers);
try {
    ResponseEntity<Void> response = this.restTemplate
            .exchange(uri, HttpMethod.POST, http, Void.class);
    if(response.getStatusCode() != HttpStatus.OK){
        throw new HttpException("Unexpected status from normalized data sending.",
                        response.getStatusCode());
    }
} catch(HttpStatusCodeException httpException){
    throw new HttpException("Failed to send normalized data.",
                        httpException.getStatusCode());
}
```

Figure 30. Microservice prototype sending data to the output system

3.3.4 Running the prototype

In a proper production environment, the running of services should be automated, for instance, via containers and systems designed for this purpose, like Kubernetes or Heroku, to name a few. However, in the scope of this prototype, services can be run individually, as building a full-scale microservice platform is not an objective of this thesis.

Since Maven is the selected project manager, it can also be used to run the developed services via the command line, calling its *mvnw spring-boot:run* command for each of the prototype's services, as shown in Figure 31 for the Data-Collector. All three services should be started separately, and once the Data- services have registered themselves to Eureka, the prototype will be ready to use via HTTP requests. To execute the whole process, a single request to the Data-Handler service has to be made, with the target source and file specified in the request payload. An example of this is shown in Figure 32, with Postman as the tool used to make the request.

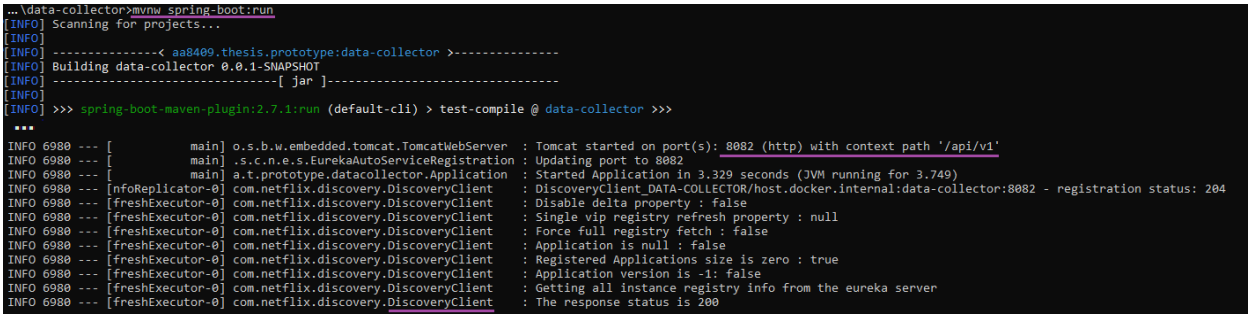


Figure 31. Running Data-Collector service via Windows Command Line with Maven

Figure 32. Microservice prototype POST to Data-Handler service

3.4 Cloud prototype implementation

The cloud prototype is a Python application running on Amazon Web Services (AWS) platform. Version 3.9 of Python is used because, at the time of the implementation, AWS did not offer support for later versions, although higher versions would have existed.

The implementation heavily uses various features offered by AWS, namely Lambda, API Gateway, Simple Notification Service (SNS), and Lambda's built-in trigger and destination components. This all results in a minimal amount of the actual Python code and can also be seen in how the prototype's structure was designed around the AWS capabilities.

3.4.1 Development environment

Cloud management was done via the AWS Management Console, a web application with individual interfaces to configure and monitor various AWS services (Amazon, 2022). The console is available at *https://console.aws.amazon.com* and requires an AWS account before it can be used (Figure 33). While Amazon does offer alternatives to the console, like extensions to popular Integration Development Environments (IDEs), they were not used during prototype implementation due to the website being robust enough.

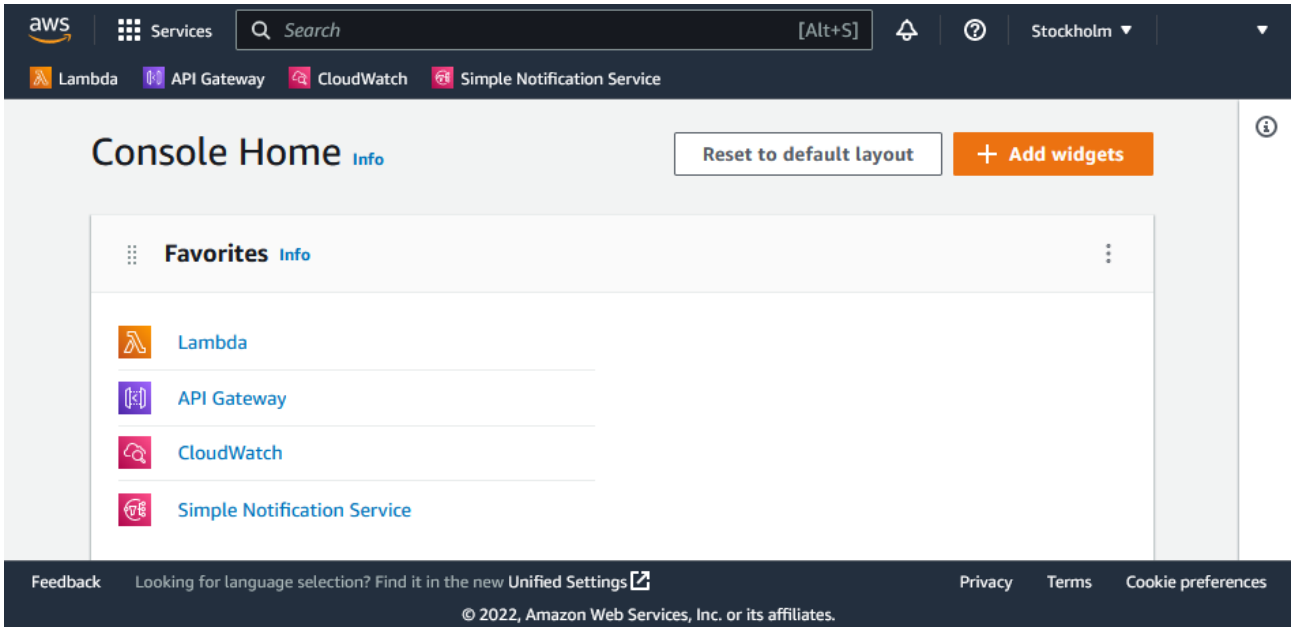


Figure 33. AWS Management Console home interface, with the shown favorites as services used in the development of the cloud prototype

AWS Management Console's Lambda section provides its own web-based editor for writing the code, although using it is not required, as the code could also be imported to and exported from Lambda. Due to the small scope of the prototype, using the editor in AWS was essentially more convenient. On the other hand, in a larger project with more complex code, a more optimal way of working would be to write the code with a more robust IDE and publish it to Lambda via CI/CD or by using one of the available AWS IDE toolkits. AWS offers a toolkit to directly access its services from various popular IDEs, like Visual Studio, Eclipse, IntelliJ, and many others (Amazon, n.d.).

3.4.2 Prototype Structure

For the main execution flow of the prototype, AWS Lambda and API Gateway services are used. Additionally, the Simple Notification Service (SNS) is used for output notification, and the Cloud-Watch is used for monitoring, although the latter two are not as integral for the prototype as the first two.

The API Gateway is used as the entry point of the execution process, receiving an HTTP request and triggering the execution of the Lambda function, with the API in question called *prototype-3-execute-API*. The execution of Lambda is done asynchronously, with the API Gateway returning a success message once the Lambda has been started, leaving the actual outcome of the process un-known at this point. This is where the SNS comes in; Lambda functions report their result to SNS; thus, the outcome of the process, be it success or failure, is notified to subscribers of the SNS. The whole execution flow is further explained in Figure 34.

Lambda has two separate functions, the *gitlab_json_source,* and *prototype_3_output*, with the former fetching and normalizing data from GitLab and the latter sending the data to the output system. These functions have no direct communication with each other; instead, AWS Lambda's *Trigger* and *Destination* features are used. The *gitlab_json_source* triggers the *prototype_3_output* once data has been successfully loaded, and the latter triggers the SNS service once data has been successfully sent.

The *Trigger* is a condition within AWS Lambda that begins the execution of a function, whereas the *Destination* is what its completion will trigger. As such, the *Destination* of the *gitlab_json_source* equals the *Trigger* of the *prototype_3_output*. There can be multiple triggers and destinations with different conditions; for instance, only successful execution of *gitlab_json_source* will trigger the output function, with failure having been configured to trigger the SNS instead.

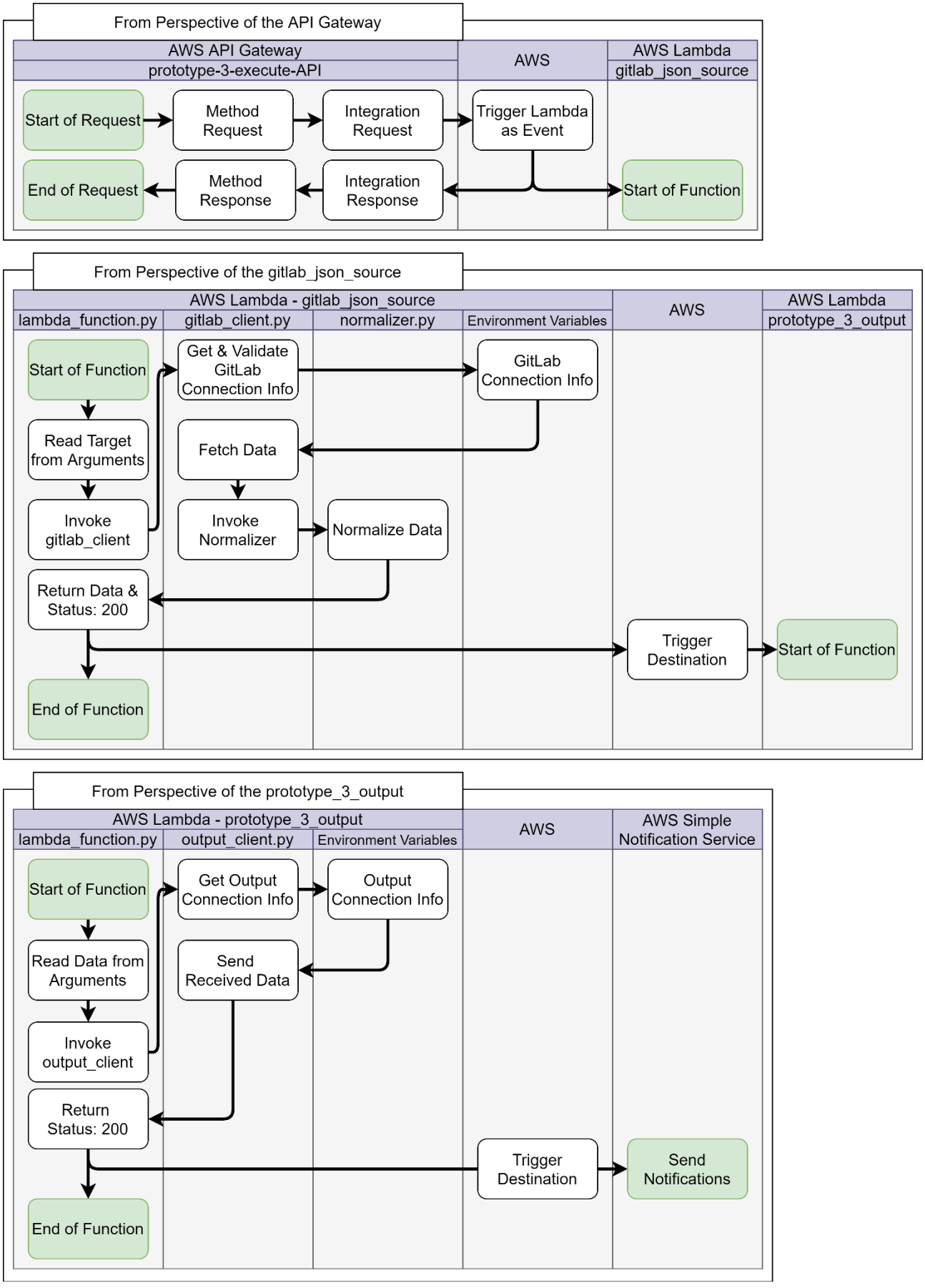


Figure 34. Cloud prototype's execution flow

Multiple endpoints can be configured for a single API of AWS, which has been used in the prototype to provide the desired functionality of having a simple way to add a new source, as shown in Figure 35. Every source has its own POST endpoint in the API, with the endpoint's trigger configured to use the respective source's Lambda function. The behavior and configuration of each Lambda function are source-specific, with all finally triggering the same output function.

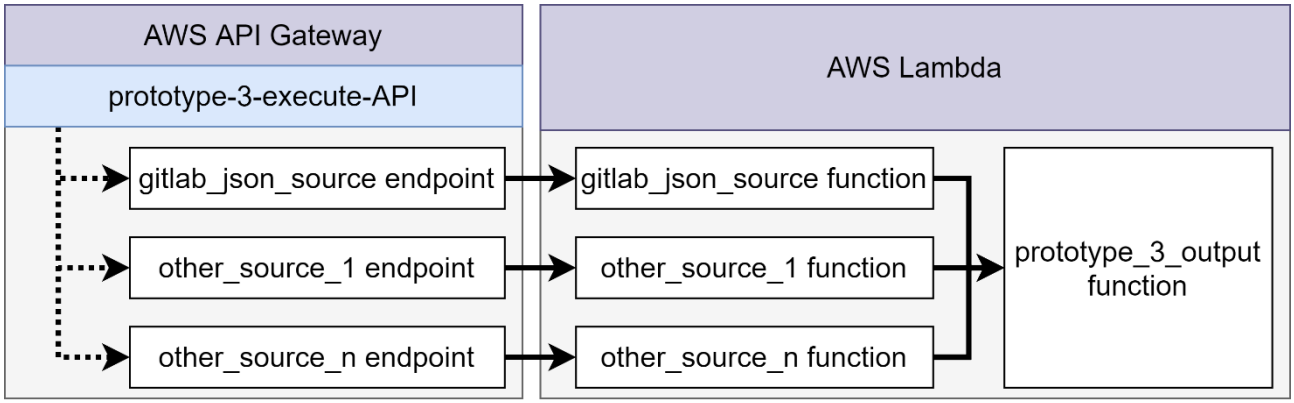


Figure 35. Cloud prototype component relation via triggers

3.4.3 Components overview

An overview of components that make up the cloud prototype, consisting of used AWS services and custom Python functions within the Lambda service.

To function together, many of AWS's components require permissions to be configured in the Management Console; however, AWS has streamlined this process by automatically suggesting and configuring these policies when attempting to set up operations that need them. These automatically given permissions can be found in the Policies section of the AWS Identity and Access Management (IAM) service, which can also be used to configure them manually. Since, in the scope of this prototype, there was no need to perform these configurations by hand, the IAM service is not part of this overview either.

AWS CloudWatch can be used to view application logs written by both AWS's internal components and *print* commands within custom Python code. These are omitted from the components overview due to not being relevant to the overall execution of the process.

API Gateway

AWS API Gateway is used to configure and expose HTTP and REST endpoints for public or internal use. When a new API is created, any number of endpoints can be registered for them.

The primary endpoint of the prototype has been named after the source it executes: *gitlab_json_source*. The actual URL of the endpoint would be like *https://UNIQUE_CODE.execute-api.AWS_REGION.amazonaws.com/default/gitlab_json_source*, where the *UNIQUE_CODE* and *AWS_REGION* are replaced with API's distinct identifier and current AWS account's selected region, respectively. A POST method has been chosen during endpoint creation, although any other regular HTTP method or a combination of them could have also been used.

Before the endpoint can be configured, both the authorization method and model for the used DTO have to be defined (Figure 36), both possible via respective tabs of the API Gateway interface. The API key is only one alternative for authentication and was selected for its simplicity, with other possibilities, such as IAM and a Lambda authorizer, also available. Model is defined as a basic JSON schema and is referred to by its given name in other parts of AWS.

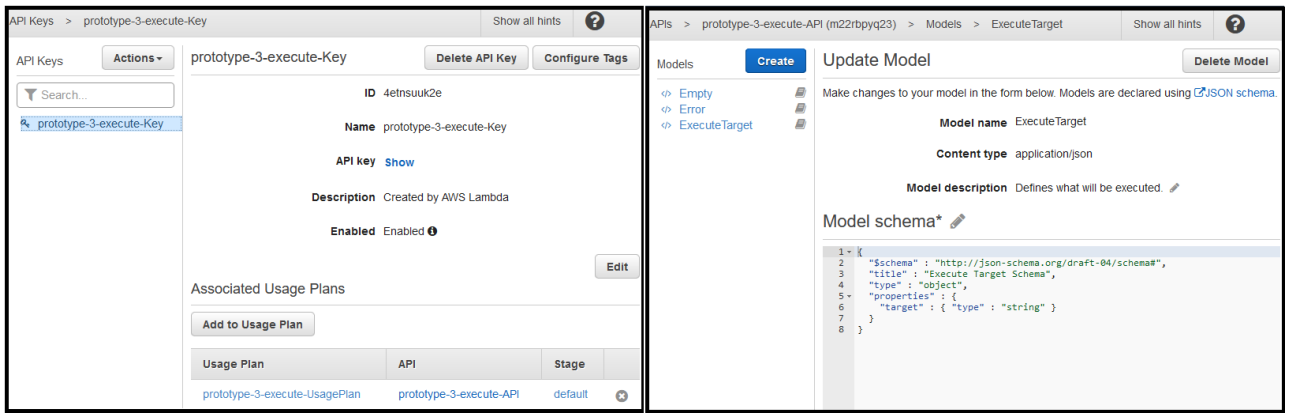


Figure 36. API Key and Model in AWS API Gateway

Configuration of the endpoint itself is divided into *Method Request*, *Integration Request*, *Integration Response*, and *Method Response*, as shown in Figure 37. However, there is no need to configure all available fields, as AWS fills most information automatically. The responses do not require

any configurations in this prototype, as the expected result is only information on whether the Lambda was triggered successfully, and default configurations provide this information.

The *Method Request* defines the expected request headers, query parameters, body, and authentication details. Only the latter two are used in this prototype, using the model and API key previously created.

The *Integration Request* defines what AWS will internally do with the received request. For this prototype, the intended action is to trigger the associated Lambda function. Additionally, due to the function execution desired as an asynchronous event, an *X-Amz-Invocation-Type* header must be registered with *'Event'* as its value, including the surrounding quotation marks.

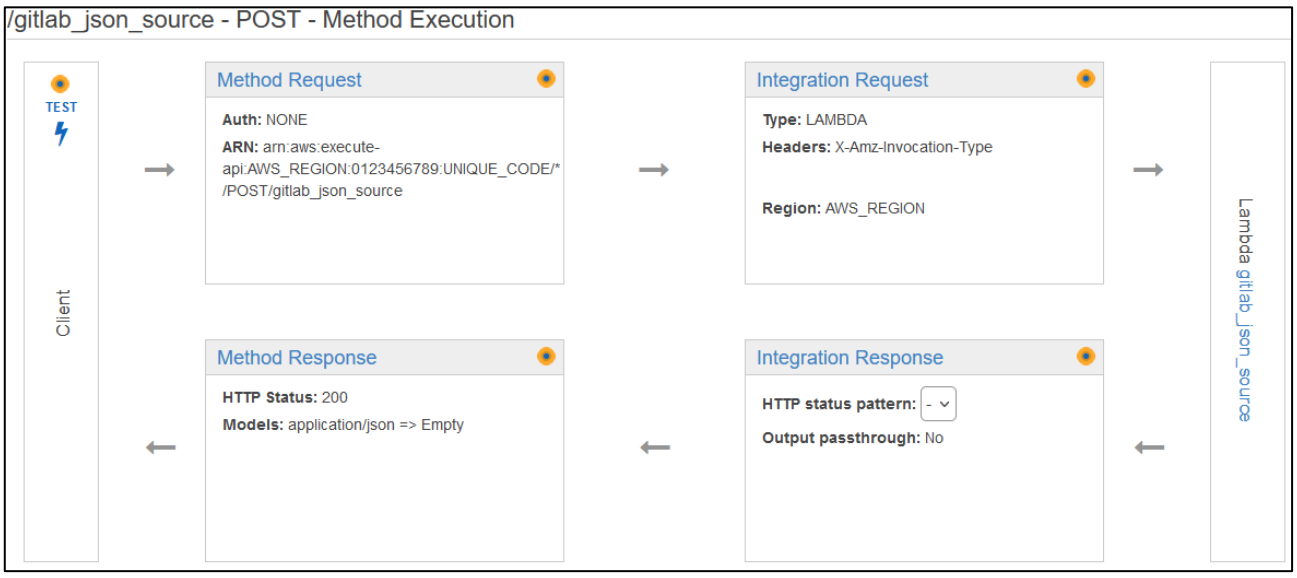


Figure 37. API Gateway endpoint configurations

Lambda configurations

Both Lambda functions require environment variables for their connection information, which is done per a Lambda function in the *Configurations* tab (Figure 38). Options for maximum allowed execution duration and memory usage can also be configured here, which are essential settings to optimize due to AWS billing the account by usage.

The same tab can also be used to define Triggers and Destinations, although the *Function overview* section offers a more visual alternative. There is no need to define the same Trigger and Destination twice; adding a Lambda function as a Destination of one will automatically add a respective Trigger to the other.

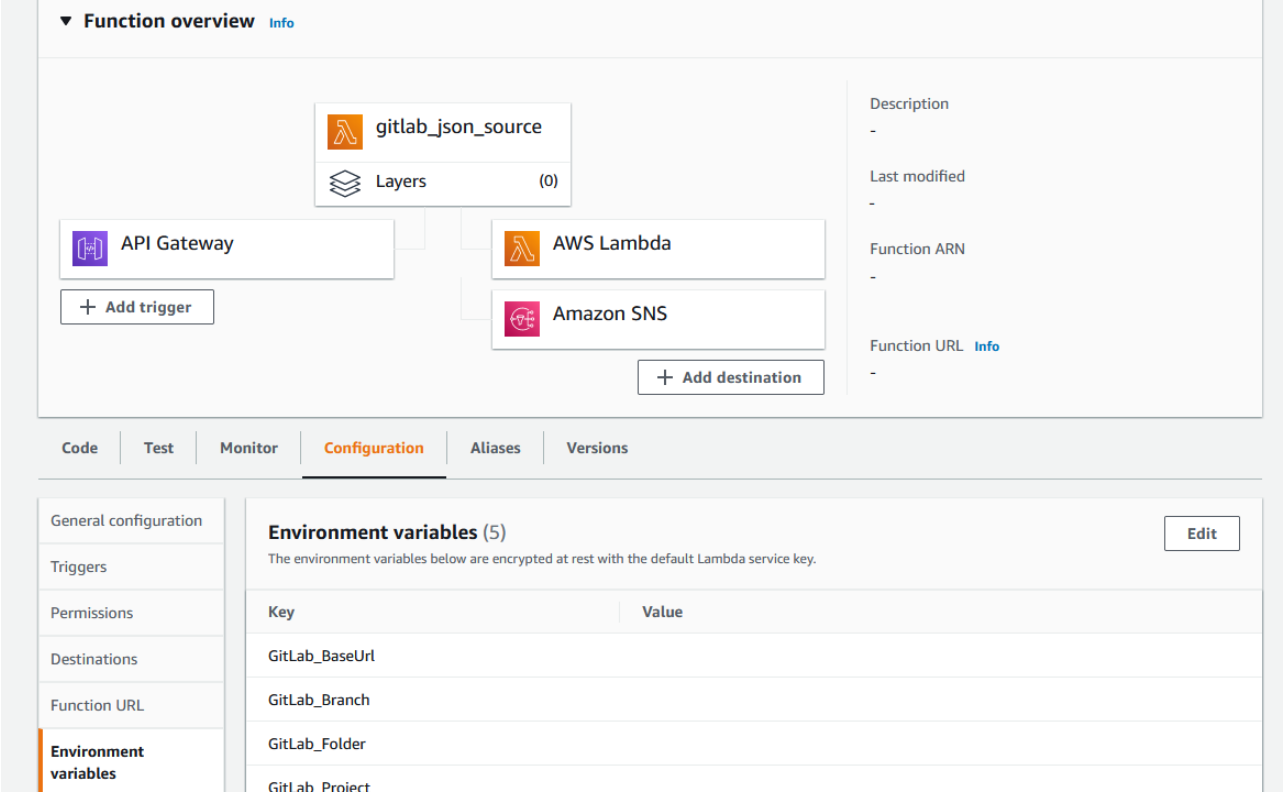


Figure 38. AWS Lambda function overview and configurations of *gitlab_json_source*

Lambda entry point

AWS Lambda has a pre-defined *lambda_function.py* file with a *lambda_handler* function that serves as the entry point of any function. These can be changed in the *Runtime settings* section of the Lambda function, although the prototype uses the defaults.

The function receives possible data from previous AWS components as arguments. Additionally, AWS provides a package called *os* for importing, containing environment information, like variables.

AWS expects the result to be of a format similar to what is shown in Figure 39, especially the *statusCode*, since the destination for success will not be triggered if the status does not indicate it. If the execution does not receive a success status or ends in a raised exception, then the failure destination will be triggered.

```
# Import that allows access to AWS Lambda Configurations:
from os import environ

# Data, like the request body can be found from the "event" argument.
def lambda_handler(event, context):
    try:
        # Actual logic will be here.
    except Exception as exc:
        raise Exception("Custom error message will be here.")

    return {
        'statusCode': 200,
        'body': 'Optional body will be here.'
    }
```

Figure 39. A generic AWS Lambda function entry point with environment imported

Since a single Lambda function can contain multiple files, it is up to the developer to decide whether to include the actual logic in the same entry file or split it into multiple ones. The prototype has been implemented with separate files, with the entry file being the only part aware of AWS-specific logic and the actual functionality to be found from different files. The benefit of this approach is that if the Python code is ever to be moved away from AWS, only the single entry file will need to be re-written or replaced.

Lambda GitLab source

The *gitlab_json_source* Lambda function consists of *lambda_function.py*, *gitlab_client.py*, and *normalizer.py* Python files. The purpose of the Lambda function is to load data from GitLab and normalize it.

The Lambda function has been configured with two possible destination triggers. If the process completes its execution successfully, a 200 response is returned from *lambda_function.py*, triggering the *prototype_3_output* function; otherwise, an error message is sent to the SNS.

The *lambda_function.py*, which is the entry point of the Lambda, reads arguments received from the AWS API Gateway and fetches environment variables from Lambda function configurations. It then calls the main function of the *gitlab_client.py* with these arguments, using the received result as the response body of the whole Lambda function. Should the data fetching fail, the original exception is merely logged, with a custom exception returned as the Lambda result.

The core functionality of *gitlab_client.py* and the whole *gitlab_json_source* is handled by its primary function: *fetch_data*. The function prepares and executes a request to GitLab, passes the received data through the *normalizer.py*, and returns the normalized data as a result of the function (Figure 40).

```
request = Request(url=full_url)
request.add_header(key="Authorization", val=f"Bearer {token}")

# Using "contextlib" to handle automatic disposing of the request.
data = None
with contextlib.closing(urlopen(request)) as response:
    data = response.read()

if not data:
    raise Exception("No data!")

data = json.loads(data.decode("utf8"))
result = list(map(normalize, iter(data)))
return result
```

Figure 40. Cloud prototype fetching data from GitLab

Lambda output

The *prototype_3_output* Lambda function consists of *lambda_function.py* and output_*client.py* Python files. The purpose of the Lambda function is to send data it receives as an argument to the output system.

The Lambda function has been configured with a single destination trigger; whether the function completes its execution successfully or fails, the response is sent to the SNS.

The entry point, *lambda_function.py,* reads normalized data passed to it as an argument, loads the output system's information from AWS Lambda configurations, and calls the *send_data* function of the *output_client.py* with these values. If the output function executes successfully, the Lambda function will return a success status. Should the output fail, the exception is caught and logged, and the Lambda function ends with a custom exception.

The *output_client.py* contains the main logic of the *prototype_3_output* and is split into two distinct functions, the primary *send_data*, which reads received Lambda Environment variables and calls the *_send_data_row* function for each received data row. The latter function forms a connection to the output system and sends all received data to it one row at a time (Figure 41). An exception is raised should the data sending fail for any reason.

```
def _send_data_row(dataRow: dict, url, token):
    body = _data_to_json_string(dataRow).encode("utf-8")
    request = Request(url=url, method="POST", data=body)
    request.add_header("Content-Type", "application/json; charset=utf-8")
    request.add_header("Content-Length", len(body))
    if token:
        request.add_header(key="Authorization", val=f"Bearer {token}")

    # Using "contextlib" to handle automatic disposing of the request.
    code = None
    with contextlib.closing(urlopen(request)) as response:
        code = response.getcode()

    if code != 200:
        raise Exception(f"Unexpected result: {code}!")
```

Figure 41. Cloud prototype sending data to output system

Simple Notification Service

The AWS Simple Notification Service, SNS for short, is used to register different messaging topics. Any amount of receivers can be subscribed to a single topic with a per-subscriber messaging protocol. Some of the supported protocols are Email, SMS, and HTTP, to name a few, with various other alternatives existing as well. Other AWS services can push messages to SNS topics, which will then cause a message to be sent to all subscribers with whichever protocol they have selected.

In the cloud prototype, an SNS topic named *prototype_3_email* is used by both Lambda functions via destination triggers. However, it would have been possible to register separate topics for both and even a different topic for the success and failure cases.

3.4.4 Running the prototype

Before the prototype can be executed, a subscription to SNS should be registered to receive a notification about the result. This can be done by selecting the prototype's topic in SNS and using the form via the *create subscription* option; an example of a filled form is shown in Figure 42. Alternatively, the progress of the prototype's process can be tracked via the *CloudWatch* service of AWS, including whether it executed successfully or not.

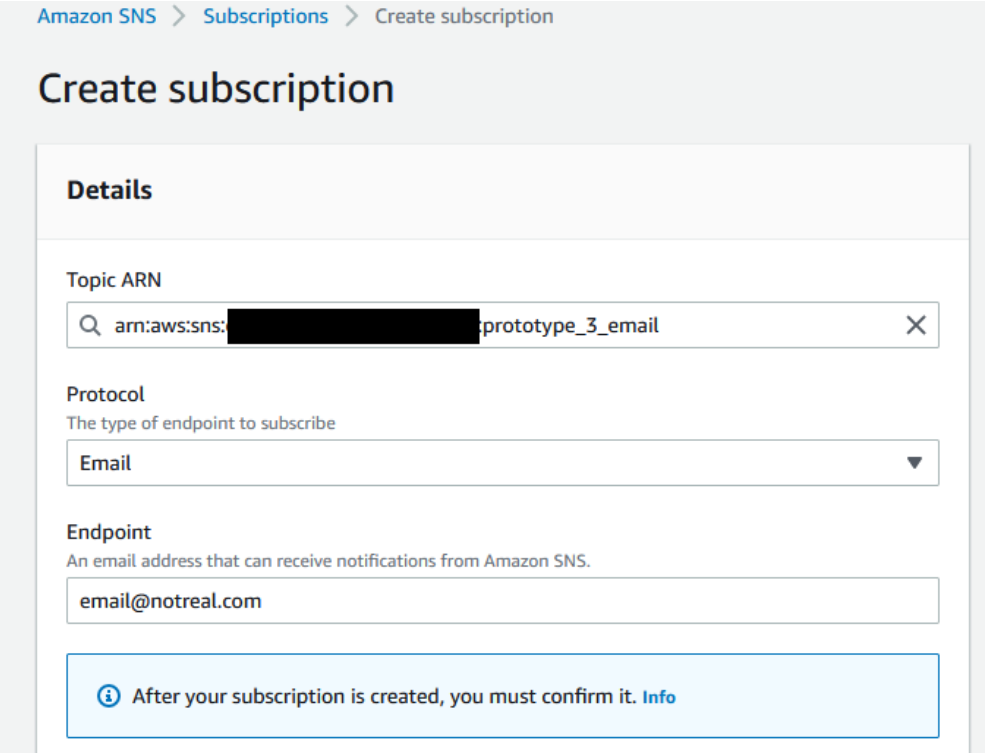


Figure 42. Subscribing to Amazon SNS

Execution of the prototype is triggered by sending a POST request to the API Gateway (Figure 43). The request requires the name of the source to use, that is, the GitLab as part of the URL path, the name of the file as a body, and the token configured for the API Gateway as a header named *X-*

API-KEY. If the endpoint is successfully triggered, a 200 response is immediately received. Later, a success or failure message should appear via the protocol used when subscribing to the SNS.

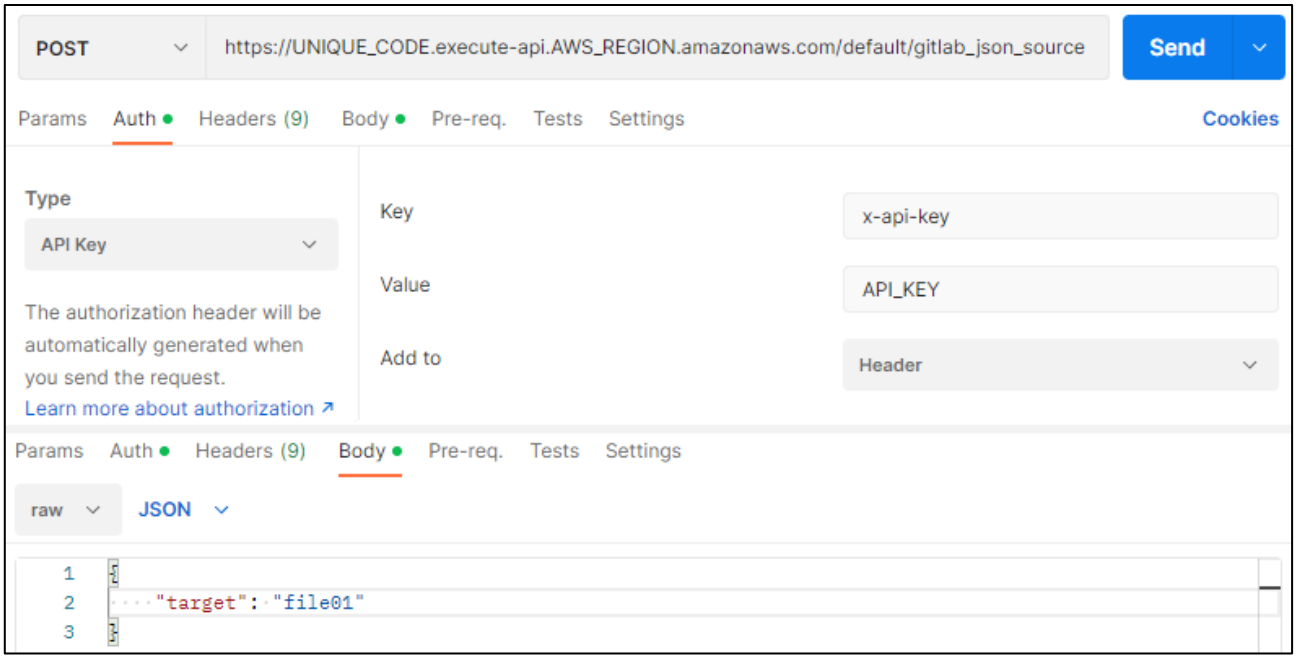


Figure 43. Cloud prototype POST to Amazon API Gateway from Postman

4 Evaluation

The purpose of the evaluation is to determine the benefit and drawbacks of each prototype's technology if selected for the development of an actual application. While used programming languages play a role in the evaluation, they are not the focus, as more weight is given to architectural models: monolith, microservice, and cloud. Each prototype is first evaluated separately within four different categories, followed by an assessment of findings and a comparison between them.

Different evaluation categories are used to analyze various aspects of a prototype:

- Observation.
- Performance.
- Relevance.
- Workload and cost.

The Observation category is qualitative, Performance and Relevance are quantitative, and Workload and cost are a mix of qualitative and quantitative.

4.1 Observation

A written assessment of prototypes from a developer's point of view, with the developer in question as the thesis author. Observations are based on notes taken during the actual implementation of each prototype and findings made from the completed versions.

Developers who build and maintain software are the ones who will be interacting with their chosen technologies the most. Multiple factors will influence a technology choice, and how a developer sees it should be one of them, improving the quality of life of developers as well as the maintainability of the software in the long run.

4.1.1 Monolithic prototype

The application itself is very simple. Everything used during implementation is basic .NET, with no need to understand advanced concepts like dependency injection, which helps lessen its learning curve and requirements should a new developer be introduced to the project. The most significant advantage of the monolithic nature is the application's shared memory and a single configuration file accessible from anywhere in the solution. This concept was the basis when designing the prototype to take full advantage of its monolithic nature.

Due to its singular nature, there is no need to design the codebase to be used by other software or for another purpose in the future, which can be seen as both a strength and weakness of monolithic architecture. Implementation can focus on the desired use case, and with the prototype requiring only marginal attention on overall design and consistency planning, its implementation is quick and efficient. However, adding new features will require developers to be aware of the application as a whole, as changing one part can easily affect another.

While the compact nature works well with the prototype, a similar approach in production will quickly lead to a solution too complex to maintain with ease. More unrelated components, all bun-

dled inside a single solution, quickly add up to quite a large codebase for a single application. Suppose any feature of the prototype is wished to be used for other purposes. In that case, it requires splitting the prototype up, copying its code to another solution, or extending it with more features that may or may not be associated with the original purpose.

4.1.2 Microservice prototype

Due to microservice architecture, a single service is dedicated to its individual purpose, a core concept used when designing the prototype. This makes designing, implementing, and maintenance of a service very straightforward. Since there is no need to directly consider or even be aware of other services and features on the shared or other service platforms, a developer can focus only on the feature they are currently dealing with.

While the prototype only has a minimal number of services, a real microservice platform will likely have more. While a single developer may only be aware of some services, a complex service platform will still require personnel aware of the platform as a whole. Otherwise, the platform can quickly become too complex, consisting of services with redundant or duplicate features, eventually negating the whole purpose of splitting features into individual services.

In the prototype, the same data passes through two different services, bringing out the main drawback of multiple services. Each service functions as its own entity, and as the count of services increases, so does the required memory usage and the amount of underlying data traffic.

Due to the individuality of services, the prototype contains a duplicate of the data transfer object (DTO) in both. This can be avoided, for instance, by having a shared DTO project, although services are only partially separate after that. Ultimately, it falls under the preference of the creator to decide whether they want fully or partially independent services within their service platform.

With its advanced concepts like Dependency Injections, Java Spring Boot has a higher learning curve than some simpler technologies; however, the implementation process becomes more straightforward for a developer already proficient with it. The framework handles most of the advanced features, like endpoints, allowing a developer to focus on the service’s intended features rather than the service itself.

4.1.3 Cloud prototype

AWS offers a large variety of highly configurable services; using them was the central concept when designing the cloud prototype. The API Gateway and SNS allowed all communication to and from the prototype to be externalized. At the same time, Lambda acts as a layer for custom business logic with only a minimum required amount of actual code.

Most of the workload during prototype implementation came from learning how to configure these services. However, the person configuring them does not have to be a software developer, making the cloud prototype's implementation very flexible. While it requires some knowledge of software development when writing the Lambda functions, most of the prototype is simply about configuring already existing components. In the end, the cloud solution moves responsibility away from developers to AWS and those skilled in configuring it.

Actual software development is simplified because each Lambda function can be treated separately, making them not unlike microservices, where a developer can focus only on a single feature at a time. Due to the most complex logic externalized to AWS, the coding of a Lambda function is possible for a developer with just a basic understanding of the chosen language.

The minimal amount of actual code also leads to the most significant problem with AWS. Because the prototype uses so many AWS features, changing to another cloud provider or away from the cloud altogether will require an equal or greater amount of work than it initially took to build in on AWS. However, this can be mitigated by moving more logic away from AWS services and into the custom Lambda functions.

4.1.4 Observation comparison

The learning curve varies by the prototype. The simplicity of monolithic means it does not require any advanced knowledge about more complex architectural models, understanding of the used programming language is enough. In contrast, the microservice requires an extensive understanding of Spring Boot and microservice architecture as a whole. The cloud prototype has the least requirements for software development skills while requiring the most knowledge about other technologies, namely AWS services.

Monolithic and microservice are contracts of each other regarding development patterns. The former is an all-in-one package, while the latter is split into as many distinct portions as possible. Development patterns of the cloud prototype cannot be as easily compared to the others, as it focuses more on configurations than actual software development.

Monolithic can be more optimal for smaller solutions than microservice unless there is an already existing and robust microservice platform to implement the new solution into. Larger-scale solutions become tricky to maintain regardless of the architectural model, although a well-maintained microservice platform will always have an advantage over a monolithic solution. With the proper configurations, the cloud can be optimized for solutions of all sizes, while not necessarily the most optimal for all cases, making it the most flexible choice.

4.2 Performance

In software development, performance evaluation is essential, as a poorly performing application is noticeable in increased memory requirements and reduced overall useability. There are multiple ways to evaluate application performance, for instance, by logging timestamps in source code, via CPU usage, or with the help of external evaluation software. Due to the small scope of prototypes, the logging approach was chosen for its simplicity and ease of implementation.

4.2.1 Logged performance

All prototypes log evaluation information on key points of their execution. Segments that should not affect evaluation are excluded; for instance, the speed of downloading data from an external system is affected by the current performance of the external system and cannot be allowed to influence performance results.

The actual logging process is done using whichever respective feature comes bundled with each technology: the .NET console output for monolithic, the default logger of Spring Boot for microservice, and the print function of Python for the cloud. Additionally, AWS logs automatically contain session information, which is used when calculating the execution time of the cloud prototype.

The performance evaluation is done by executing each prototype with differently sized data files and collecting logged timestamps. Each prototype is run ten times per target file, and the timestamp used in the evaluation is their calculated mean value. The used data source is the GitLab source that has been described per prototype in the Prototypes section, with the data files used as:

1. File01 of 797 bytes.
2. File02 of 7940 bytes.
3. File03 of 79006 bytes.

The performance of all prototypes is shown in Table 1, split between the execution of own code and built-in segments of the technology. The table does not include time spent on external actions, like requests to GitLab.

Table 1. The performance of prototypes in milliseconds

Prototype	Data	Own code			Built-in			Total		
		Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Monolithic	File01	277	288	281	72	85	77	351	367	358
	File02	297	335	307	73	90	80	376	425	388
	File03	497	556	518	75	81	78	578	633	596
Microservice	File01	14	20	17	98	113	104	117	127	122
	File02	32	52	41	94	112	104	132	159	145
	File03	200	272	233	93	103	97	299	365	330
Cloud	File01	10	46	25	361	489	430	391	520	454
	File02	126	215	178	388	477	445	586	668	624
	File03	1606	1995	1707	441	1138	592	2047	2808	2299

Test cases with results inconsistent with the overall performance are left out of the evaluation. For instance, 9 out of 10 executions of File02 for the monolithic prototype are between 376 and 425 milliseconds, while the remaining execution is 1611 milliseconds and thus left out of the evaluation. Additionally, tests with File01 and File02 of the cloud prototype have a single aberration each. There are no irregular test results for the microservice prototype.

As the execution process of the Cloud prototype differs from others by relying more heavily on built-in segments of AWS, the time spent between triggering of API Gateway and the beginning of Lambda and the transition between Lambda functions is counted as built-in duration.

4.2.2 Performance comparison

When comparing prototypes with each other, logged performance alone is not comparable, as the execution environment will have a considerable effect on durations. As such, the performance evaluation does not focus on actual execution times but on overall performance. This is achieved via two different cases:

1. Examining what portion of total execution is used by own code and what amount is used by technology's built-in segments.
2. Measuring how increasing the size of the processed data affects the overall performance.

The percentage of total execution spent on own code is shown in Figure 44, using mean values of each test case. Figure 45 displays how many percentages duration of File02 and File03 is longer than the execution of File01.

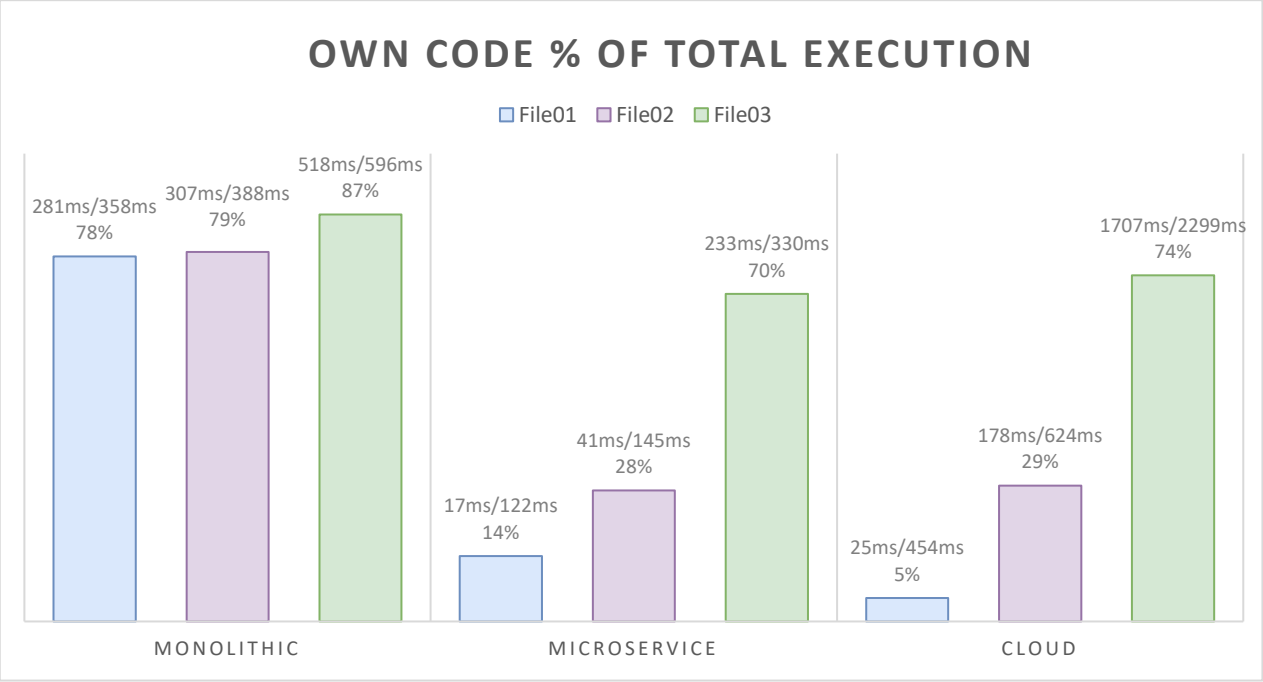


Figure 44. Division of prototypes execution time between own and built-in code

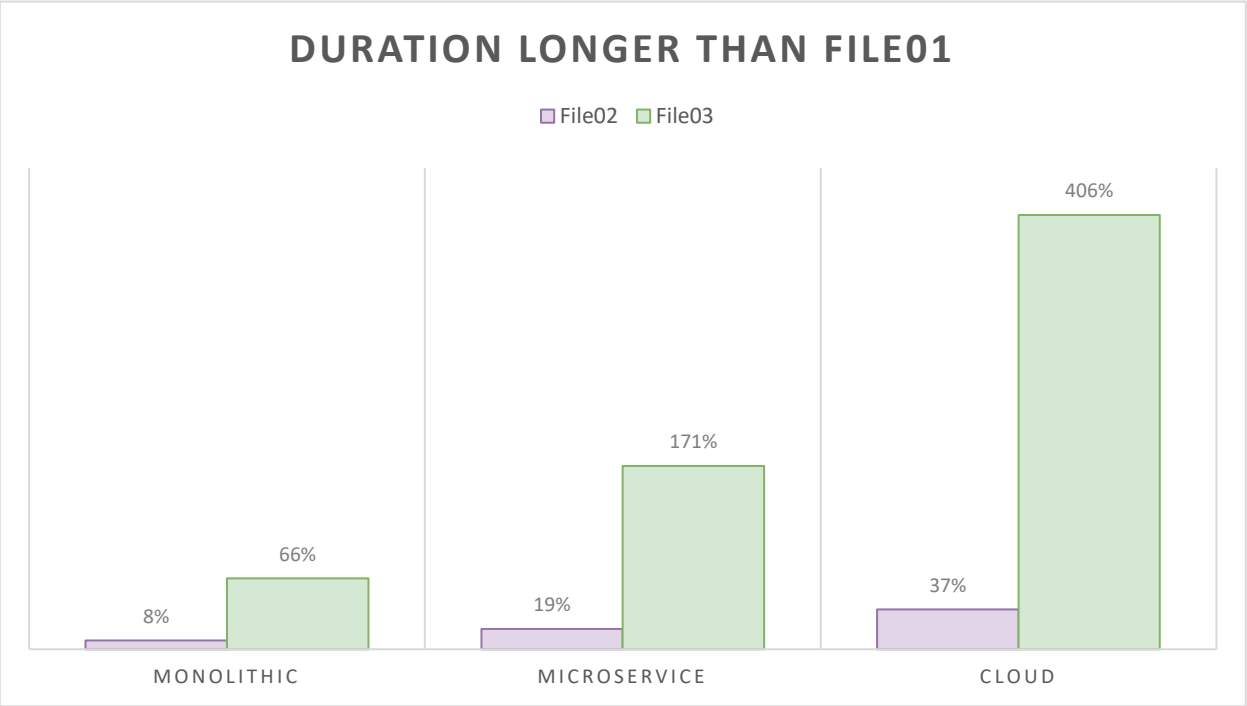


Figure 45. File02 and File03 durations in relation to File01

The monolithic prototype has the highest overall time spent on its custom code. The explanation is that it is the simplest of all prototypes. Unlike other prototypes, it does not use any advanced framework or execution flow; apart from initialization and teardown of the main application, all logic is done by self-written code.

For the microservice and cloud prototypes, the amount spent on their code is minimal when dealing with the smaller files, as both prototypes have externalized their most complicated logic to the underlying framework. Time spent in the own code increases with the file size, while the size is barely noticeable during the execution of built-in segments.

The duration of the cloud prototype increases drastically with a larger file. AWS has various configurations available for Lambda functions and other services, allowing memory usage and other factors affecting the overall performance to be further configured. As such, allocating more resources to the prototype would have increased its performance, although due to the AWS pricing model, this will also increase the monthly bill.

Total duration will always increase when the size of a file to process grows; that is unavoidable. However, when Figure 44 and Figure 45 are compared, it becomes apparent that the amount of time spent in the own code is affected the most. The difference between durations in Figure 45 is the least noticeable on the monolithic because, as shown in Figure 44, it has the highest amount of time spent within its code to begin with. The more drastic duration differences seen with microservice and cloud are synchronous with the incrementation in the time spent on their own code.

With microservice and cloud prototypes, some of their increment in duration can be attributed to additional data transfer and memory requirements, as opposed to the monolithic running the whole process within a single memory thread. However, since the bulk of the increment in all prototypes is still due to the own code, this emphasizes the need for proper code optimization rather than reflecting on the architectural models themselves.

4.3 Relevance

As software development techniques improve, some older technologies are replaced by new alternatives, while others remain popular. Relevance is evaluated by first defining the core technologies of prototypes, then finding out how many open job positions are looking for a developer with knowledge of them.

4.3.1 Job Postings

The location of job postings is narrowed down to Finland, with the advertised position as Programmer, Software Developer, Full Stack Developer, or one of the Finnish alternatives. The search was done in September of 2022, including all open positions found at that date.

LinkedIn was chosen as the job posting site to search from due to its popularity in Finland and globally. When browsing LinkedIn postings, a common trend among some companies with offices in multiple cities is to create a different posting for each. Because these are, in truth, the same job position, all postings with identical descriptions are treated as one during the evaluation. After removing duplicates, the total number of found job postings is 851.

Due to LinkedIn's extensive amount of available job postings, the search was not done by hand; instead, it was done via automated queries. Multiple variations of each technology's spelling were defined and used to find matches in LinkedIn postings. The downside is that while computerized logic accurately collects the data, connecting it to prototypes' technologies can be expected to have some margin of error due to rare variations of keywords and possible spelling mistakes.

4.3.2 Relevance comparison

Keywords of used technologies and architectures, excluding alternative spellings and Finnish translations, are presented in Table 2, and Figure 46 shows which percentage of all job postings contain these keywords. The evaluation based on this search does not accept partially matching words to reduce possible false positives. For instance, the .NET has to either be one of the alternative formats, like ASP.NET, or the .NET as a single word; otherwise, even a simple web address ending in .net would be considered a match.

Table 2. Relevance keywords

	Technology Keywords	**Architecture Keywords**
Monolithic prototype	.NET, C#, console application	Monolithic
Microservice prototype	Spring Boot, Java, Eureka, service registry, Maven, POM	Microservice, service platform
Cloud prototype	AWS (including explicit service names), Python	Cloud, serverless

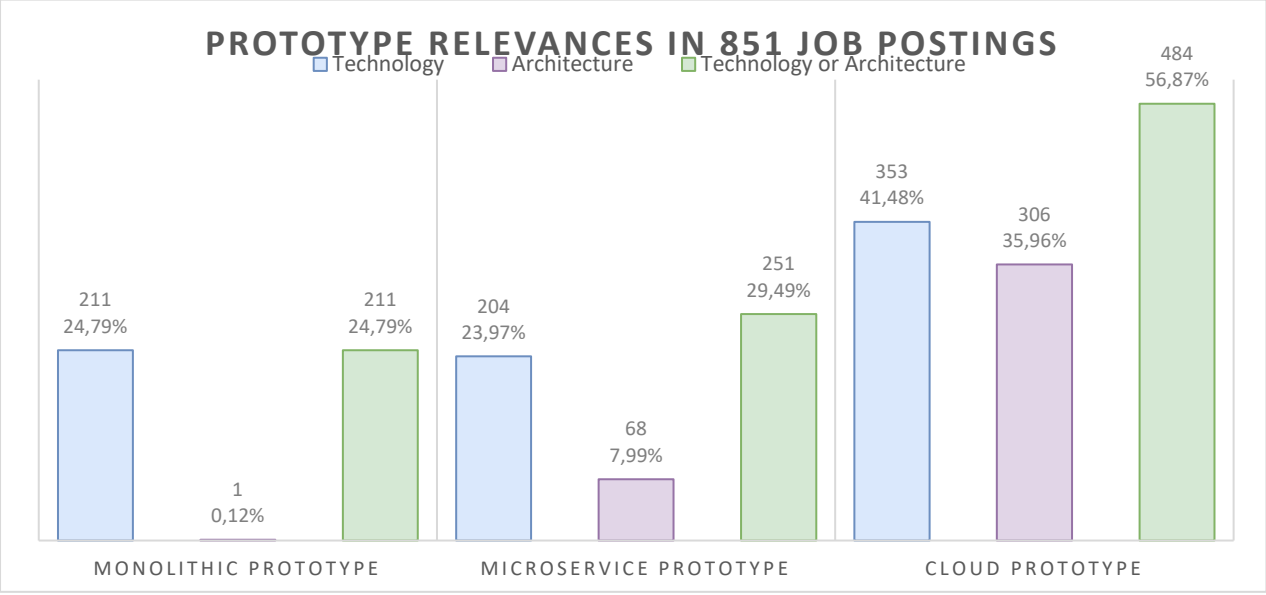


Figure 46. Prototype relevances in 851 job postings

The monolithic architecture has only a single match, which can be explained by the somewhat negative association with the term. There could be multiple companies with monolithic solutions; they simply do not advertise them. Even the single match announces that the company is shifting from monolithic to microservices, although maintaining solutions is listed as a responsibility of the searched developer; thus, this match is counted as both microservice and monolithic, as it can be assumed that some of the old monolithic solutions are part of the maintenance responsibilities.

The popularity of the cloud prototype stands out from the results. It does not come as a real surprise that almost half of the job postings mention either AWS or Python, as both were chosen for the prototype because of their popularity. Since architectures are not matched with any specific technology in mind, the potential matches count increases, as can be seen especially with the cloud, as the term cloud encompasses a wider variety of concepts; for instance, both cloud-based development and cloud-based hosting appear in the search.

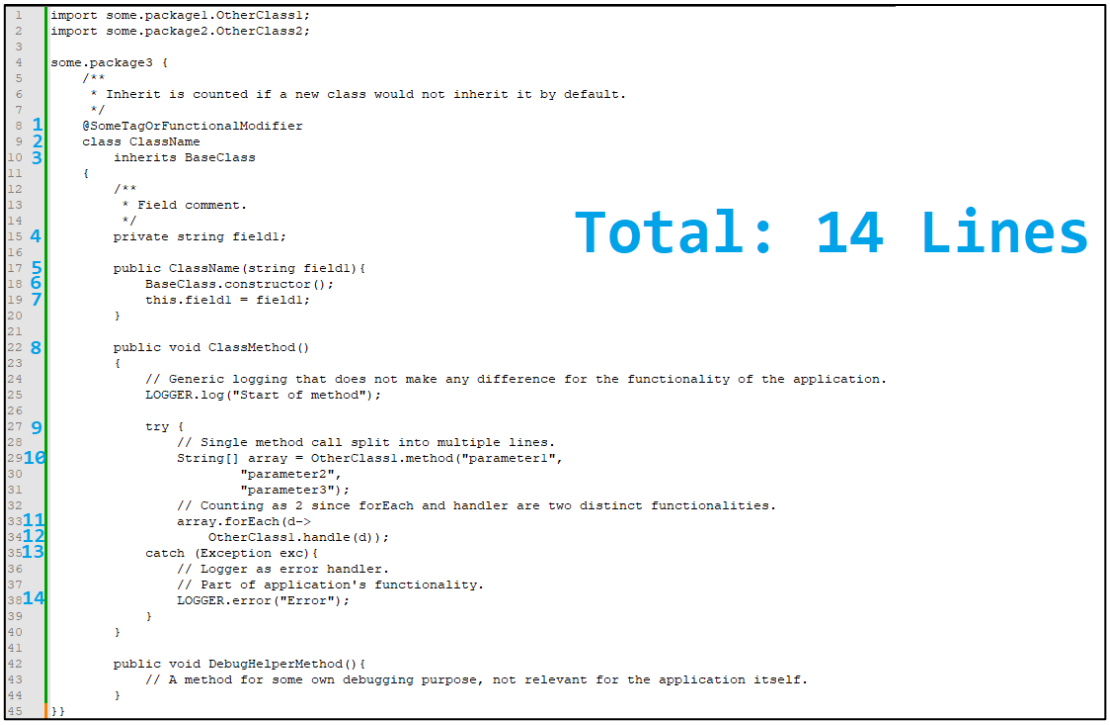
Results of the relevance search show similar trends between technologies, although different percentages, than similar studies; for instance, a study by DevJobScanner.com shows 20% for Python, 17% for Java, and 12% for C# (Logan, 2022). This difference can be explained by the more constricted search pool and the focus on prototypes as a whole instead of an individual programming language that represents it.

4.4 Workload and cost

A less technical aspect to consider is how much developing and maintaining software will cost. This evaluation assesses the cost of possible licenses, other fees of selected technologies, and the workload required from an employee developing and maintaining it. The workload is measured by the total lines of code used to implement each prototype, indicating the amount of required initial development work and maintenance involvement. A more complex codebase means that software will also be more challenging to maintain.

When calculating the code lines, only functional code is included, and lines split into multiple rows for readability are counted as one. All metadata information is ignored, for instance, comments, imports, empty lines, and brackets. Additionally, any line or file not part of the original prototype

specifications is skipped; for instance, non-functional logging and testing-related lines. A concrete example of which lines are counted is demonstrated via pseudo-code in Figure 47.




```
import some.package1.OtherClass1;
import some.package2.OtherClass2;

some.package3 {
    /**
     * Inherit is counted if a new class would not inherit it by default.
     */
1   @SomeTagOrFunctionalModifier
2   class ClassName
3       inherits BaseClass
    {
        /**
         * Field comment.
         */
4       private string field1;

5       public ClassName(string field1){
6           BaseClass.constructor();
7           this.field1 = field1;
        }

8       public void ClassMethod()
        {
            // Generic logging that does not make any difference for the functionality of the application.
            LOGGER.log("Start of method");

9           try {
                // Single method call split into multiple lines.
10              String[] array = OtherClass1.method("parameter1",
                        "parameter2",
                        "parameter3");
                // Counting as 2 since forEach and handler are two distinct functionalities.
11              array.forEach(d->
12                  OtherClass1.handle(d));
13          catch (Exception exc){
                // Logger as error handler.
                // Part of application's functionality.
14              LOGGER.error("Error");
            }
        }

        public void DebugHelperMethod(){
            // A method for some own debugging purpose, not relevant for the application itself.
        }
}}
```

Figure 47. Counting code lines from a pseudo-code example

4.4.1 Workload comparison

The total lines of code per prototype are 194 for the monolithic, 407 for the microservice, and 73 for the cloud, as further illustrated in Figure 48. Design choices, developer preferences, and language-specific patterns will affect the actual lines of code. As such, lines of codes from prototypes are only approximate, and a single line should not hold too much weight in the overall assessment.

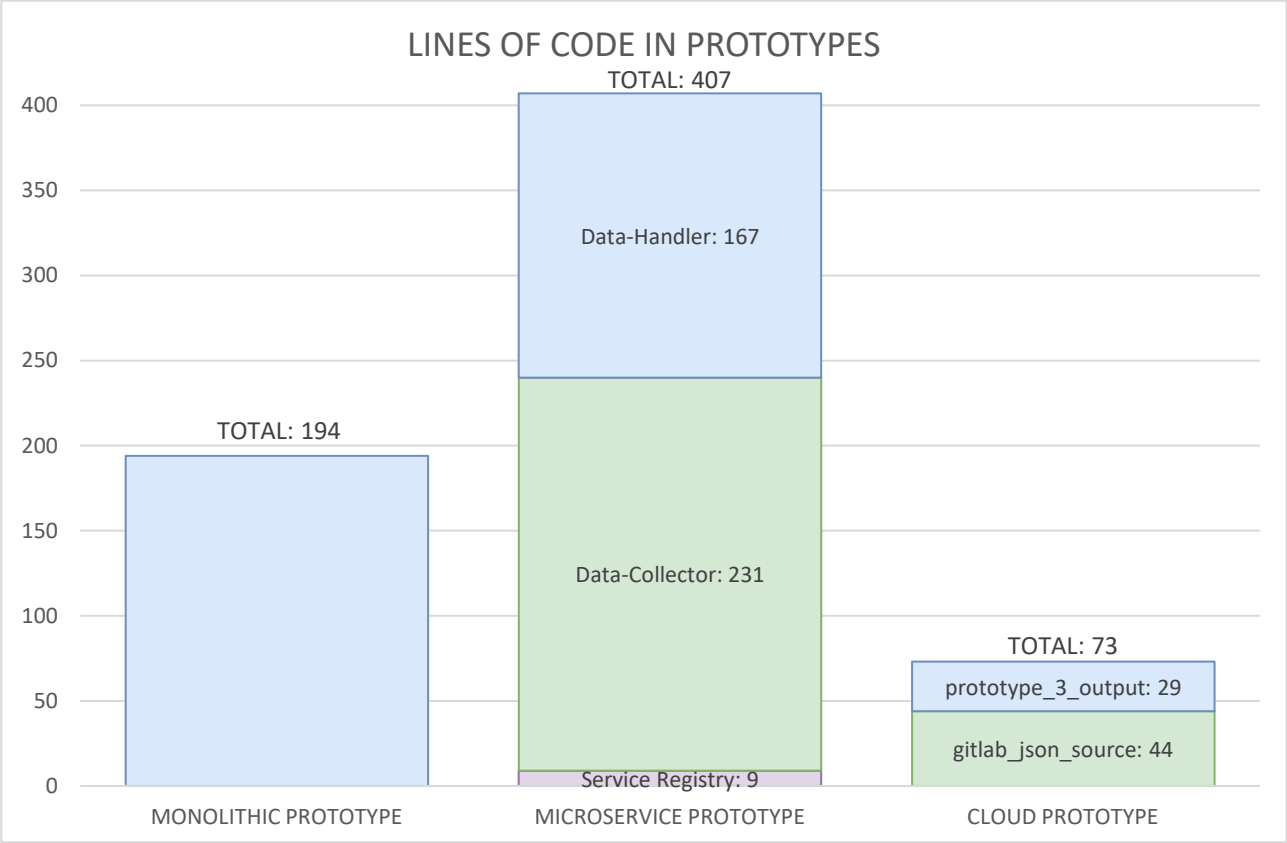


Figure 48. Lines of code in prototypes

The Microservice prototype has the highest amount of code. The amount is affected by both the selected technology of Java Spring Boot and Microservice architecture. As Java syntax requires defining private fields with getter and setter methods, this will quickly increase the total amount of code. While there are solutions for streamlining this, like Project Lombok, the purpose of the prototype is not to find solutions to archaic implementation patterns and thus does not make use of them. The amount of code is also incremented by the requirement for each microservice to be fully independent, thus replicating DTO classes in both services. A shared DTO project would reduce the amount of code in the long run.

The cloud prototype has the least amount of code. This is because it heavily uses available AWS services and built-in features instead of relying too much on its own code. While this does reduce the workload for both initial implementation and post-release support, it does not consider that

AWS services still have to be configured and maintained by someone. However, the AWS Management Console's usability makes this process relatively straightforward and less time-consuming than maintaining the complete solution.

4.4.2 Cost of prototypes

While the workload evaluation offers an overview of how much development and maintenance will cost as required work by an employee, there are also additional costs to consider.

Hosting

Only the cloud prototype comes with built-in application hosting. The monolithic and microservice prototypes must be hosted by purchasing a hosting environment from a third party or self-hosting within a custom server. While self-hosting may require the initial purchase of the physical server, the main cost comes from having an employee on payroll to configure the hosting environment and maintain it, which could also be the case with a third-party environment, depending on the scope of services offered.

Monolithic prototype

The monolithic prototype uses Visual Studio and .NET, which has multiple pricing options based on the company's size and whether the application is for commercial use. Different editions and their pricing offered by Microsoft in October of 2022 are illustrated in Table 3 and Table 4, respectively. While it is possible to choose software other than Visual Studio or find different prices from alternative vendors, evaluating them is not part of this Thesis.

Table 3. Visual Studio editions (Microsoft, 2022d)

Edition	Organization limitations
Community	Max 5 potential users. Non-enterprise organization. Software developed under an open license.
Professional	Enterprise organization with <= 250 PCs and <=$1 million in annual revenue.
Enterprise	Enterprise organization with > 250 PCs or > $1 million in annual revenue.

Table 4. Visual Studio pricing per one user (Microsoft, 2022d)

Edition	Subscription	Monthly Cost ($)	Yearly Cost ($)	Billing Period
Community	Free	0	0	Free
Professional	Cloud	45	540	Monthly
	Standard	100	1 199	Yearly, first year
	Standard	67	799	Yearly, renewal
	Standalone License	499	499	One-time payment
Enterprise	Cloud	250	3 000	Monthly
	Standard	500	5 999	Yearly, first year
	Standard	214	2 569	Yearly, renewal

The main difference between cloud and standard subscriptions is the more generous billing period of the cloud, although the cloud lacks subscriber benefits, which include additional software and enhanced support and training that come with standard licenses (Microsoft, 2022d). Higher versions of Visual Studio come with more features, except the standalone license, which does not include updated Visual Studio versions, Azure DevOps features, or Subscriber benefits.

These costs make it apparent that larger organizations and those desiring additional benefits of the Enterprise version will pay a lot more per license than others. Cheaper licenses are still viable if developers have no acute need for extended features, in which case cloud and standalone Professional editions are the most feasible option.

Microservice prototype

Technologies used with the microservice prototype have no explicit fees specific to them. Cost is the workload of development, maintenance, and chosen hosting environment.

Cloud prototype

AWS is billed by usage. While a free tier exists for individual use, it is not limitless or available for all organizations. AWS Management Console has a Pricing Calculator service, which can be used to estimate combined prices from various services. Different services have different pricing factors; the API Gateway is invoiced per request, while the Lambda is billed by execution duration.

As the size of requests increases the execution duration of the cloud prototype, Table 5 contains cost estimates for a single request using the files used during the Performance evaluation. The evaluation ignores free tier benefits and focuses on the core AWS Services used with the prototype: AWS API Gateway and Lambda. Additional costs can be expected, for instance, from the SNS service execution output, which can be given any amount of message receiver subscribers.

Table 5. Cloud prototype cost of a single request

	API Gateway ($)	Lambda gitlab_json_source ($)	Lambda prototype_3_output ($)
file01	0,0000035000	0,0000007438	0,0000037229
file02	0,0000035000	0,0000007979	0,0000355272
file03	0,0000035000	0,0000009333	0,0003504570

While the cost of a single request appears cheap, a factor not included in these calculations is that the prototype is, just as its name suggests, a representation of a small segment of an application. Multiple factors will increase the cost in the long run; as the application becomes more complex, so does its duration increase. Since the example cost only represents a single request, every additional execution will increment the total cost by the same amount.

Due to being billed by duration, the quality of the custom Lambda code will also become a factor. Poorly written code will be heavier and may take longer to execute, thus directly increasing costs.

4.4.3 Cost comparison

Regarding costs of development and upkeep of software, Java Spring Boot is a perfect choice when an organization wants to avoid extra licenses, provided that there are developers to maintain it and the organization has access to a proper hosting environment. Most of the cost comes from employees developing and maintaining the service platform. Solutions from all prototypes require employees, although the microservice has the highest workload, which correlates to potentially higher employee costs.

If licenses are not an issue, AWS is a cheap alternative, especially for solutions that are either small, rarely executed, or properly optimized. Although the by-usage model of costs can quickly become surprisingly high if not well monitored.

The opposite of AWS is the .NET and Visual Studio; with its higher license fees, it is not an optimal choice for such a solution unless the company also has other software they develop with it. Overall, whether Visual Studio is worth it depends on if the additional features bundled with its subscriptions are useful to the organization and to what extent the C# and .NET are used; after all, it is the preferred IDE for .NET development.

5 Conclusions

Answers to the original research questions of *"What are the strengths and weaknesses of each selected technology?"* and *"How do the selected technologies compare with each other?"* can be summarized based on gathered and evaluated data from prototypes and findings from the literature review.

5.1 Summary of findings

There are various technologies and architectural models to choose from. While this thesis only deals with three variations, it became apparent that there is no single way to decide which is the best. Much depends on the software itself, how large it is, who develops it, and how it shall be hosted, amongst other possible variables.

The monolithic is the simplest solution to implement, as long as its scope remains small and purpose singular, quickly becoming harder to create and a burden to maintain if the application grows too complex. A passable, although rather quick and dirty, choice for cases where anything more complicated could be considered an overkill or development resources are limited.

The microservice approach offers great re-usability and a more focused development process as long as the service platform does not get out of hand. A solid choice if its more demanding development and maintenance workload, including hosting an entire service platform, is not an issue.

The cloud prototype is a low-code serverless solution that mostly requires understanding the AWS itself. Less dependent on software developers and self-hosting, AWS offers minimized development effort and externalized responsibilities. A very flexible choice when committing to AWS in

the long term is not a problem, and applications are compact enough to avoid excessive usage costs.

The monolithic architecture represents the most traditional and bare-bones software development, while the microservice architecture improves upon it. Although both are still relevant in modern software development, the popularity of the cloud is on the growth. If solutions that outsource responsibilities away from developers keep gaining popularity, it will change how software development is approached and what will be expected from software developers in the future.

5.2 Reflection on research and development

Implemented prototypes and their evaluation provided answers for the main research questions of the thesis. On the other hand, two software are rarely alike and different outcomes, both better and worse, could have been reached with other design choices.

Prototyping is an effective data collection method for software evaluation as it allows the simulation of real applications with minimal work by focusing on the aspect to evaluate. Additionally, designing and implementing the prototypes helps to hone one's overall capabilities as a software developer. However, while defining requirements for prototypes, their scope could have been greater. While focusing on the primary use case allowed more precise evaluation, it did not allow technologies to be evaluated at their full potential. Even though made with different architectural models and technologies, the prototypes are still quite similar at their core. As such, having larger prototypes with additional features would have enriched the overall depth and accuracy of the evaluation.

Defining what is expected of each architecture model, as is done in the literature review section, is helpful when designing prototypes to reflect them accurately. However, comparing the implemented prototypes to their definition reveals that not all aspects were considered due to the smaller scope of the overall research. This is especially the case with the microservice prototype since a considerable part of microservices is their environment, including the business side and development patterns, which could not be considered within the scope of this thesis.

Cost evaluation offered a good overview, although it was primarily theoretical. Many additional variables will affect the costs of actual software development. For instance, there could be agreements between companies to reduce prices, and the skill levels of individual employees will affect development times and quality.

5.3 Further research suggestions

Prototypes used in the thesis only represent a few architecture and technology combinations. Similar comparisons could be made with other variations, even within the selected technologies. For instance, a monolithic software running on the cloud or a modular monolith combining microservice and monolithic architectures.

Prototypes focused on the development of the primary use case. Further research could focus more on the CI/CD aspect, considering how each prototype becomes a fully released product. Due to differences between the three environments, each will consist of steps and configurations that differ from the others.

Testing automation is an integral part of any software development. However, it was omitted entirely from this research, as its scope is vast enough to serve as a topic of a whole new thesis.

References

Agamez, S. (n.d.). *Modular Monolithic vs. Microservices*. Retrieved from FullStack Labs: https://www.fullstacklabs.co/blog/modular-monolithic-vs-microservices

Amazon. (2022). *What is the AWS Management Console?* Retrieved from AWS: https://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/learn-whats-new.html

Amazon. (n.d.). *Tools to Build on AWS*. Retrieved from AWS: https://aws.amazon.com/developer/tools/

Ashtari, H. (2022, April 5). *What Are Microservices? Definition, Examples, Architecture, and Best Practices for 2022*. Retrieved from Spiceworks: https://www.spiceworks.com/tech/devops/articles/what-are-microservices/

Chandrakant, K. (2022, December 17). *Introduction to Serverless Architecture*. Retrieved from Baeldung: https://www.baeldung.com/cs/serverless-architecture

Datadog. (2021, June 28). *Serverless Architecture Overview*. Retrieved from Datadog: https://www.datadoghq.com/knowledge-center/serverless-architecture/

Datadog. (2022, June). *The state of serverless*. Retrieved from Datadog: https://www.datadoghq.com/state-of-serverless/

Evaluation: What is it and why do it? (n.d.). Retrieved from meera: https://meera.snre.umich.edu/evaluation-what-it-and-why-do-it

Foote, K. D. (2021, April 22). *A Brief History of Microservices*. Retrieved from Dataversity: https://www.dataversity.net/a-brief-history-of-microservices

Fowler, M. (2014, March 25). *Microservices*. Retrieved from https://martinfowler.com/articles/microservices.html

Ganeshchowdharysadanala. (2021a, July 11). *Spring Boot – Code Structure*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/spring-boot-code-structure/

Ganeshchowdharysadanala. (2021b, July 11). *Spring Initializr*. Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/spring-initializr/

IBM Cloud Education. (2021, March 30). *Microservices*. Retrieved from IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/microservices

InterviewBit. (2022, August 26). *Spring vs Spring Boot: Know The Difference*. Retrieved from InterviewBit: https://www.interviewbit.com/blog/spring-vs-spring-boot/

Kanjilal, J. (2020, January 24). *Pros and cons of monolithic vs. microservices architecture*. Retrieved from TechTarget: https://www.techtarget.com/searchapparchitecture/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture

King, B. (2022, January 2). *What is FaaS? Function as a Service explaine*. Retrieved from DigitalOcean: https://www.digitalocean.com/blog/what-is-faas-function-as-a-service-explained

Law, M. (2023, February 15). *Top 10 biggest cloud providers in the world in 2023*. Retrieved from Technology Magazine: https://technologymagazine.com/top10/top-10-biggest-cloud-providers-in-the-world-in-2023

Logan. (2022, December 12). *Top 8 Most Demanded Programming Languages in 2022*. Retrieved from devjobsscanner: https://www.devjobsscanner.com/blog/top-8-most-demanded-languages-in-2022/

Long, J. (2015, January 20). *Microservice Registration and Discovery with Spring Cloud and Netflix's Eureka*. Retrieved from spring.io: https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka

Maayan, I. (2022, August 3). *Will Modular Monolith Replace Microservices Architecture?* Retrieved from Medium: https://medium.com/att-israel/will-modular-monolith-replace-microservices-architecture-a8356674e2ea

Mendonça, N. C., Box, C., Manolache, C., & Ryan, L. (2021, October). *The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture*. doi:https://doi.ieeecomputersociety.org/10.1109/MS.2021.3080335

Microsoft. (2021, December 16). *Monolithic applications*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications

Microsoft. (2022a, October 28). *Visual Studio Community*. Retrieved from Microsoft: https://visualstudio.microsoft.com/vs/community/

Microsoft. (2022b, April 10). *.NET vs. .NET Framework for server apps*. Retrieved from Microsoft: https://learn.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server

Microsoft. (2022c, November 2). *Why did we build Visual Studio Code?* Retrieved from Visual Studio Code: https://code.visualstudio.com/docs/editor/whyvscode

Microsoft. (2022d, October 14). *Visual Studio Pricing*. Retrieved from Microsoft Visual Studio: https://visualstudio.microsoft.com/vs/pricing-details/

MW Team. (2023, January 6). *What are Microservices? How Microservices architecture works*. Retrieved from Middleware: https://middleware.io/blog/microservices-architecture/

Newtonsoft. (n.d.). *Performance Tips*. Retrieved from newtonsoft: https://www.newtonsoft.com/json/help/html/Performance.htm

Pletcher, S. (2022, May 18). *AWS vs Azure vs GCP: The big 3 cloud providers compared*. Retrieved from A Cloud Guru: https://acloudguru.com/blog/engineering/aws-vs-azure-vs-gcp-the-big-3-cloud-providers-compared

Quilliam, E. (2022, February 25). *What are the Different Software Prototyping Methods?* Retrieved from itenterprise: https://itenterprise.co.uk/software-prototyping-methods/

Red Hat. (2022, August 19). *What is cloud architecture?* Retrieved from Red Hat: https://www.redhat.com/en/topics/cloud-computing/what-is-cloud-architecture

Richardson, C. (n.d.). *Pattern: Messaging*. Retrieved from microservices.io: https://microservices.io/patterns/communication-style/messaging.html

vanshika4042. (2022, August 2). *7 Best Java IDE For Developers in 2022*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/7-best-java-ide-for-developers-in-2022/

VMWare Tanzu. (n.d.). *Service Registration and Discovery*. Retrieved from spring.io: https://spring.io/guides/gs/service-registration-and-discovery/

VMware. (n.d.). *What is Cloud Architecture?* Retrieved from vmware: https://www.vmware.com/topics/glossary/content/cloud-architecture.html

Wadhwani, P., & Loomba, S. (2021, June 16). *Serverless Architecture Market size worth $30 Bn by 2027*. Retrieved from Global Market Insights: https://www.gminsights.com/pressrelease/serverless-architecture-market

Appendices

Appendix 1. Monolithic prototype IntegrationManagerService

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 1.

using System.Collections.Generic;
using System.Threading.Tasks;
using ThesisPrototypeNet.Business.Output;
using ThesisPrototypeNet.Business.Source;
using ThesisPrototypeNet.Model;
using ThesisPrototypeNet.Model.Enums;
using ThesisPrototypeNet.Model.Interfaces;

namespace ThesisPrototypeNet.Business
{
    /// <summary>
    /// Service that handles execution of the data integration logic.
    /// </summary>
    public class IntegrationManagerService
    {
        /// <summary>Create new intergation manager service.</summary>
        public IntegrationManagerService()
        {
            //
        }

        /// <summary>Execute the integration process.</summary>
        /// <param name="sourceID">
        /// <see cref="IDataSource.ID"/> to integrate data from.
        /// </param>
        /// <param name="targetData">
        /// Defines the target data at the <see cref="IDataSource"/> to integrate.
        /// </param>
        public async Task ExecuteAsync(string sourceID, DataTarget targetData)
        {
            // Importing data from source system.
            IDataSource source = SourceProvider.Instance.Get(sourceID);
            List<NormalizedData> data = await source.FetchDataAsync(targetData);

            // Exporting data to output system.
            OutputClient outputAccess =
                        new OutputClient(OutputConnectionInfo.FromConfig());
            await outputAccess.SendAsync(data);
        }
    }
}
```

Appendix 2. Monolithic prototype SourceProvider

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 1.

using System.Collections.Generic;
using System.Linq;
using ThesisPrototypeNet.Business.Source.GitLabJson;
using ThesisPrototypeNet.Business.Source.HardCoded;
using ThesisPrototypeNet.Model.Exceptions;
using ThesisPrototypeNet.Model.Interfaces;

namespace ThesisPrototypeNet.Business.Source
{
    /// <summary>Provider for <see cref="IDataSource"/> instances.</summary>
    public class SourceProvider
    {
        /// <summary>Singleton instance of this provider.</summary>
        private static SourceProvider instance;

        /// <summary>Gets the singleton instance of this provider.</summary>
        public static SourceProvider Instance
        {
            get
            {
                if (instance == null)
                {
                    instance = new SourceProvider();
                    instance.Init();
                }
                return instance;
            }
        }

        /// <summary>Instances of registered data sources.</summary>
        private List<IDataSource> dataSources;

        /// <summary>
        /// Create new data source provider.
        /// Prefer using the <see cref="Instance"/> instead.
        /// </summary>
        protected SourceProvider()
        {
            this.dataSources = new List<IDataSource>();
        }

        /// <summary>
        /// Initialize registered instances of <see cref="IDataSource"/>.
        /// </summary>
        protected void Init()
        {
            this.dataSources.Clear();
            this.dataSources
                .Add(new GitLabJsonSource(GitLabConnectionInfo.FromConfig()));
        }
```

```
        /// <summary>
        /// Initialize registered instances of <see cref="IDataSource"/>.
        /// </summary>
        protected void Init()
        {
            this.dataSources.Clear();
            this.dataSources
                .Add(new GitLabJsonSource(GitLabConnectionInfo.FromConfig()));
        }

        /// <summary>
        /// Get the instance of a data source that matches
        /// given arguments.
        /// </summary>
        /// <param name="id">
        /// <see cref="IDataSource.ID"/> of the source to get.
        /// </param>
        /// <returns>Existing instance of the data source.</returns>
        /// <exception cref="DataSourceNotFoundException">
        /// Thrown when no data source was found with
        /// given <paramref name="id"/>.
        /// </exception>
        public IDataSource Get(string id)
        {
            IDataSource result = this.dataSources
                .FirstOrDefault(source => string.Equals(source.ID, id,
                         System.StringComparison.OrdinalIgnoreCase));
            if (result != null)
            {
                return result;
            }

            throw new DataSourceNotFoundException(
                        "Did not find data source with given ID.", id);
        }
    }
}
```

Appendix 3. Monolithic prototype GitlabJsonSource

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 1.

using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using ThesisPrototypeNet.Model;
using ThesisPrototypeNet.Model.Exceptions;
using ThesisPrototypeNet.Model.Interfaces;

namespace ThesisPrototypeNet.Business.Source.GitLabJson
{
    /// <summary>Data source of GitLab json.</summary>
    public class GitLabJsonSource
        : IDataSource
    {
        /// <summary>Unique identifier of this source.</summary>
        public const string SourceID = "GitLabJson";

        /// <summary>
        /// Format for fetching a JSON file from GitLab, using V4 API.
        /// </summary>
        private const string API_V4_JSON_FORMAT
            = "{0}/api/v4/projects/{1}/repository/files/{2}{3}.json/raw?ref={4}";

        /// <summary>The epoch/unix-0 date and time.</summary>
        private static readonly DateTime EPOCH
            = new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);

        /// <summary>Defines the connection to use.</summary>
        private GitLabConnectionInfo connectionInfo;

        /// <summary>Create new data source.</summary>
        /// <param name="connectionInfo">Defines the connection to use.</param>
        public GitLabJsonSource(GitLabConnectionInfo connectionInfo)
        {
            this.connectionInfo = connectionInfo;
        }

        /// <summary>Gets identifier that distinguishes this source.</summary>
        public string ID { get => SourceID; }

        /// <summary>
        /// Set authorization header of given <paramref name="request"/>,
        /// if authorization is available.
        /// </summary>
        /// <param name="request">Request whose authorization to set.</param>
        private void SetAuthorization(HttpRequestMessage request)
        {
            if (!string.IsNullOrWhiteSpace(this.connectionInfo.Token))
                request.Headers.Authorization
                    = new AuthenticationHeaderValue("Bearer",
                                                    this.connectionInfo.Token);
        }
```

```
        /// <summary>Fetch and normalize data from this source.</summary>
        /// <param name="target">
        /// Determines which of the available data to fetch.
        /// </param>
        /// <returns>List of found data. Empty list if none is found.</returns>
        public async Task<List<NormalizedData>> FetchDataAsync(DataTarget target)
        {
            try
            {
                List<GitLabJsonData> result = null;
                using (HttpClient client = new HttpClient())
                using (HttpRequestMessage request =
                    new HttpRequestMessage(HttpMethod.Get, this.GetUrl(target)))
                {
                    // Authorization header from config.
                    this.SetAuthorization(request);
                    using (HttpResponseMessage response
                        = await client.SendAsync(request))
                    {
                        response.EnsureSuccessStatusCode();
                        string responseBody = await response.Content
                                    .ReadAsStringAsync();
                        result = JsonConvert
                          .DeserializeObject<List<GitLabJsonData>>(responseBody);
                    }
                }
                return result.Select(this.NormalizeData).ToList();
            }
            catch (Exception exception)
            {
                throw new DataLoadingException("Failed to fetch data from "
                    + this.GetType().Name, this.ID, target, exception);
            }
        }

        /// <summary>
        /// Get url to fetch data from.
        /// </summary>
        /// <param name="target">Defines which data to fetch.</param>
        /// <returns>Full url to fetch data from.</returns>
        private string GetUrl(DataTarget target)
        {
            return string.Format(API_V4_JSON_FORMAT,
                this.connectionInfo.BaseUrl,
                this.connectionInfo.Project,
                this.connectionInfo.Folder,
                target.TargetName,
                this.connectionInfo.Branch);
        }

        /// <summary>Normalize given data.</summary>
        /// <param name="sourceData">Data to normalize.</param>
        /// <returns>Normalized data.</returns>
        private NormalizedData NormalizeData(GitLabJsonData sourceData)
        {
            NormalizedData result = new NormalizedData();
            result.Name = sourceData.Category + " " + sourceData.Type;
            result.Value = sourceData.Value ?? 0.0;
            if (sourceData.Value.HasValue)
                result.Timestamp
                    = EPOCH.AddMilliseconds(sourceData.Value.Value);
            return result;
        }
    }
}
```

Appendix 4. Monolithic prototype OutputClient

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 1.

using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Threading.Tasks;
using ThesisPrototypeNet.Model;
using ThesisPrototypeNet.Model.Exceptions;

namespace ThesisPrototypeNet.Business.Output
{
    /// <summary>Data output connection.</summary>
    public class OutputClient
    {
        /// <summary>Defines the connection to use.</summary>
        private OutputConnectionInfo connectionInfo;

        /// <summary>Create new output access.</summary>
        /// <param name="connectionInfo">
        /// Defines the connection to use.
        /// </param>
        public OutputClient(OutputConnectionInfo connectionInfo)
        {
            this.connectionInfo = connectionInfo;
        }

        /// <summary>
        /// Send given <paramref name="dataList"/> to the output system.
        /// </summary>
        /// <param name="dataList">Data to send.</param>
        /// <exception cref="DataSendingException">
        /// Thrown when sending fails.
        /// </exception>
        public async Task SendAsync(List<NormalizedData> dataList)
        {
            try
            {
                using (HttpClient client = new HttpClient())
                    for (int index = 0; index < dataList.Count; index++)
                    {
                        NormalizedData data = dataList[index];
                        await this.SendAsync(dataList[index], client);
                    }
            }
            catch (Exception exception)
            {
                if (exception is DataSendingException)
                    throw;
                throw new DataSendingException(
                    "Failed to send data to output system.", exception);
            }
        }
```

```
        /// <summary>Send given data row to the output system.</summary>
        /// <param name="data">Data to send.</param>
        /// <param name="client">Open client connection to send with.</param>
        /// <exception cref="DataSendingException">
        /// Thrown when sending fails.
        /// </exception>
        private async Task SendAsync(NormalizedData data, HttpClient client)
        {
            using (HttpRequestMessage request =
                        new HttpRequestMessage(HttpMethod.Post,
                                               this.connectionInfo.Url))
            {
                // Authorization header from config.
                this.SetAuthorization(request);

                // The JsonContent with default settings will create a JSON
                // with output system's expected data format from
                // the NormalizedData.
                request.Content = JsonContent.Create(data);

                using (HttpResponseMessage response =
                       await client.SendAsync(request))
                {
                    if (response.StatusCode != HttpStatusCode.OK)
                    {
                        string responseBody = await response.Content
                            .ReadAsStringAsync();
                        throw new DataSendingException(
                            responseBody, response.StatusCode);
                    }
                }
            }
        }

        /// <summary>
        /// Set authorization header of given <paramref name="request"/>,
        /// if authorization is available.
        /// </summary>
        /// <param name="request">Request whose authorization to set.</param>
        private void SetAuthorization(HttpRequestMessage request)
        {
            if (!string.IsNullOrWhiteSpace(this.connectionInfo.Token))
                request.Headers.Authorization =
                    new AuthenticationHeaderValue("Bearer",
                                this.connectionInfo.Token);
        }
    }
}
```

Appendix 5. Microservice prototype DataService of Data Collector service

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 2.

package aa8409.thesis.prototype.datacollector.services;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import aa8409.thesis.prototype.datacollector.models.DataSource;
import aa8409.thesis.prototype.datacollector.models.NormalizedData;
import aa8409.thesis.prototype.datacollector.sources.DataSourceProvider;
/**
 * Service for fetching of normalized data.
 */
@Service
public class DataService {
    /**
     * Provider used to determine which data source to use.
     */
    private final DataSourceProvider dataSourceProvider;
    /**
     * Create new service.
     * @param dataSourceProvider Provider used to determine which data source to
     *                           use.
     */
    public DataService(@Autowired DataSourceProvider dataSourceProvider) {
        this.dataSourceProvider = dataSourceProvider;
    }
    /**
     * Fetch data from given source.
     * @param sourceID   Unique identifier of the data
     *                   source to fetch the data
     *                   from.
     * @param targetName Defines the target data to fetch
     *                   from the source.
     * @return List of data. Throws exception if data fetcing fails.
     */
    public List<NormalizedData> get(String sourceID, String targetName) {
        DataSource source = this.dataSourceProvider.get(sourceID);
        List<NormalizedData> result = source.fetchData(targetName);
        return result;
    }
}
```

Appendix 6. Microservice prototype DataSourceProvider of Data Collector service

```java
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 2.

package aa8409.thesis.prototype.datacollector.sources;

import java.util.List;
import org.apache.commons.lang.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import aa8409.thesis.prototype.datacollector.models.DataSource;
import aa8409.thesis.prototype.datacollector.models.NotFoundException;

/**
 * Provider for {@link DataSource}s.
 */
@Component
public class DataSourceProvider {
    /**
     * All sources that this provider can provide.
     */
    private final List<DataSource> sources;
    /**
     * Create a new provider.
     *
     * @param sources All sources that can be provided.
     */
    public DataSourceProvider(@Autowired List<DataSource> sources) {
        this.sources = sources;
    }
    /**
     * Get data source with given id.
     *
     * @param sourceID {@link DataSource#getID()}
     * @return Found data source.
     *         Throws {@link NotFoundException} when not found.
     */
    public DataSource get(String sourceID) {
        return this.sources.stream()
                .filter(d -> StringUtils.equals(d.getID(), sourceID))
                .findFirst()
                .orElseThrow(() ->
                new NotFoundException("Data Source", sourceID));
    }
}
```

Appendix 7. Microservice prototype GitlabJsonSource of Data Collector service

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 2.

package aa8409.thesis.prototype.datacollector.sources.gitlab;

import java.net.URI;
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.List;
import java.util.stream.Collectors;
import org.apache.commons.lang.ObjectUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.core.env.Environment;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;
import aa8409.thesis.prototype.datacollector.models.DataLoadingException;
import aa8409.thesis.prototype.datacollector.models.DataSource;
import aa8409.thesis.prototype.datacollector.models.NormalizedData;

/**
 * Data source of GitLab json.
 */
@Component
public class GitlabJsonSource implements DataSource {
    /**
     * The unique identifier of this source.
     */
    public static final String ID = "GitLabJson";
    /**
     * Format for fetching a JSON file from GitLab,
     * using V4 API.
     */
    private static final String API_V4_JSON_FORMAT
    = "%1$s/api/v4/projects/%2$s/repository/files/%3$s%4$s.json/raw?ref=%5$s";
    /**
     * UTC-0 timezone.
     */
    private static final ZoneId UTC0 = ZoneId.of("UTC");
```

```
    /**
     * Type reference for response reading.
     */
    private static final ParameterizedTypeReference<List<GitLabJsonData>>
    TYPE_REFERENCE = new ParameterizedTypeReference<List<GitLabJsonData>>() {
    };
    /**
     * The current environment.
     */
    private final Environment environment;
    /**
     * The rest template that this source will use for connections.
     */
    private final RestTemplate restTemplate;
    /**
     * Create new source.
     *
     * @param environment  The current environment.
     * @param restTemplate The rest template that this source will use for
     *                     connections.
     */
    public GitlabJsonSource(@Autowired Environment environment,
                @Autowired RestTemplate restTemplate) {
        this.environment = environment;
        this.restTemplate = restTemplate;
    }
    /**
     * Gets unique identifier of this data source.
     *
     * @return ID.
     */
    public String getID() {
        return ID;
    }
    /**
     * Build URI to fetch data from.
     *
     * @param connectionInfo Connection info to build from.
     * @param target         Target file to fetch.
     * @return New URI.
     */
    private URI getUri(GitLabConnectionInfo connectionInfo, String target) {
        return URI.create(String.format(API_V4_JSON_FORMAT,
                connectionInfo.getBaseUrl(),
                connectionInfo.getProject(),
                connectionInfo.getFolder(),
                target,
                connectionInfo.getBranch()));
    }
```

```
    /**
     * Fetch data.
     * @param targetName Defines the target data to fetch.
     * @return List of data. Throws exception if data fetcing fails.
     */
    public List<NormalizedData> fetchData(String target) {
        // Preparing the connection information.
        GitLabConnectionInfo connectionInfo
            = GitLabConnectionInfo.fromEnvironment(this.environment);
        URI uri = this.getUri(connectionInfo, target);
        HttpHeaders headers = new HttpHeaders();
        headers.setBearerAuth(connectionInfo.getToken());
        HttpEntity<?> http = new HttpEntity<>(headers);

        List<GitLabJsonData> responseBody = null;
        try {
            // Fetching the data. Throws exception if fails.
            ResponseEntity<List<GitLabJsonData>> response = this.restTemplate
               .exchange(uri, HttpMethod.GET, http, TYPE_REFERENCE);
            responseBody = response.getBody();
        } catch (Exception exception) {
            throw new DataLoadingException(
                    exception.getMessage(), this.getID(), target);
        }
        if (responseBody == null) {
            throw new DataLoadingException(
                    "Response body is null.", this.getID(), target);
        }
        // Returning the normalized variant of the data.
        return responseBody.stream()
                .map(this::normalizeData)
                .collect(Collectors.toList());
    }
    /**
     * Convert given GitLab data format to normalized.
     * @param sourceData Data to convert.
     * @return Normalized data.
     */
    private NormalizedData normalizeData(GitLabJsonData sourceData) {
        NormalizedData result = new NormalizedData();
        result.setName(sourceData.getCategory() + " " + sourceData.getType());
        result.setValue((double) ObjectUtils.
                        defaultIfNull(sourceData.getValue(), 0.0));
        if (sourceData.getValue() != null) {
            Instant instant = Instant.ofEpochMilli(sourceData.getValueOn());
            result.setTimestamp(LocalDateTime.ofInstant(instant, UTC0));
        }
        return result;
    }}
```

Appendix 8. Microservice prototype HadlerService of Data Handler service

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 2.
package aa8409.thesis.prototype.datahandler.services;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import aa8409.thesis.prototype.datahandler.clients.DataCollectorClient;
import aa8409.thesis.prototype.datahandler.clients.DataOutputClient;
import aa8409.thesis.prototype.datahandler.models.ExecutionTarget;
import aa8409.thesis.prototype.datahandler.models.NormalizedData;
/** Service for data handling.*/
@Service
public class HandlerService {
    /**
     * Client used to access data-collector service.
     */
    private final DataCollectorClient dataCollectorClient;
    /**
     * Client used to send data to output system.
     */
    private final DataOutputClient outputClient;
    /**
     * Create a new service.
     * @param dataCollectorClient Client used to access
     *                            data-collector service with.
     * @param outputClient        Client used to send data to output system.
     */
    public HandlerService(@Autowired DataCollectorClient dataCollectorClient,
            @Autowired DataOutputClient outputClient) {
        this.dataCollectorClient = dataCollectorClient;
        this.outputClient = outputClient;
    }
    /**
     * Handle data of the given target.
     * @param target Defines the data to handle.
     * @return OK when success.
     */
    public void handle(ExecutionTarget target) {
        List<NormalizedData> data = this.dataCollectorClient
            .getNormalizedData(target.getSourceID(), target.getTargetName());
        for (int index = 0; index < data.size(); index++) {
            this.outputClient.send(data.get(index));
        }
    }}
```

Appendix 9. Microservice prototype DataCollectorClient of Data Handler service

```
// Copyright (c) AA8409@student.jamk.fi 2022.
// This file is part of Thesis Prototype 2.
package aa8409.thesis.prototype.datahandler.clients;

import java.net.URI;
import java.util.List;
import org.apache.commons.lang.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.core.env.Environment;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpStatusCodeException;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;
import aa8409.thesis.prototype.datahandler.models.HttpException;
import aa8409.thesis.prototype.datahandler.models.NormalizedData;
import aa8409.thesis.prototype.datahandler.models.ServiceNotFoundException;
/** Client for data-collector service access. */
@Component
public class DataCollectorClient {
    /**
     * Key of the environment property containing name of the data-collector
     * service.
     */
    private static final String COLLECTOR_SERVICE_NAME_PROPERTY
    = "aa8409.thesis.prototype.datahandler.collectorServiceName";
    /**
     * Type reference for response reading.
     */
    private static final ParameterizedTypeReference<List<NormalizedData>>
     TYPE_REFERENCE = new ParameterizedTypeReference<List<NormalizedData>>() {};
    /**
     * Client to discover other services with.
     */
    private final DiscoveryClient discoveryClient;
    /**
     * The current environment.
     */
    private final Environment environment;
```

```
    /**
     * The current rest template.
     */
    private final RestTemplate restTemplate;

    /**
     * Create new service.
     *
     * @param discoveryClient Client to discover other services with.
     * @param environment     The current environment.
     * @param restTemplate    The current rest template.
     */
    public DataCollectorClient(@Autowired DiscoveryClient discoveryClient,
            @Autowired Environment environment,
            @Autowired RestTemplate restTemplate) {
        this.discoveryClient = discoveryClient;
        this.environment = environment;
        this.restTemplate = restTemplate;
    }

    /**
     * Fetch normalized data from given source.
     *
     * @param sourceID   Unique identifier of the data source to fetch the data
     *                   from.
     * @param targetName Defines the target data to fetch from the source.
     * @return List of data. Error status code if could not fetch data.
     */
    public List<NormalizedData> getNormalizedData(String sourceID,
                       String targetName) {
        URI uri = getDataFetchUri(sourceID, targetName);
        HttpHeaders headers = new HttpHeaders();
        HttpEntity<?> http = new HttpEntity<>(headers);
        try {
            ResponseEntity<List<NormalizedData>> response = this.restTemplate
                    .exchange(uri, HttpMethod.GET, http, TYPE_REFERENCE);
            return response.getBody();
        } catch (HttpStatusCodeException httpException) {
            throw new HttpException(
                    "Failed to fetch normalized data.",
                       httpException.getStatusCode());
        }
    }
```

```
    /**
     * Get URI of the data-collector service's data fetching endpoint.
     *
     * @param sourceID   Unique identifier of the data source to fetch the data
     *                   from.
     * @param targetName Defines the target data to fetch from the source.
     * @return Built URI, or throws {@link ServiceNotFoundException}
     * if not found.
     */
    private URI getDataFetchUri(String sourceID, String targetName) {
        return UriComponentsBuilder
                .fromUri(this.getRootUri())
                .pathSegment("api", "v1", "data")
                .queryParam("source_id", sourceID)
                .queryParam("target_name", targetName)
                .encode()
                .build().toUri();
    }

    /**
     * Get root url of the data-collector service.
     *
     * @return Found URI, or throws {@link ServiceNotFoundException}
     *  if not found.
     */
    private URI getRootUri() {
        String name = this.environment
            .getProperty(COLLECTOR_SERVICE_NAME_PROPERTY);
        if (StringUtils.isBlank(name)) {
            throw new IllegalStateException(
                    "Service is missing required configuration: " +
                            COLLECTOR_SERVICE_NAME_PROPERTY);
        }
        List<ServiceInstance> instances = this.discoveryClient
            .getInstances(name);
        if (instances.size() == 0) {
            throw new ServiceNotFoundException(
                    "Service with given name was not found!", name);
        }
        return instances.get(0).getUri();
    }
}
```

Appendix 10. Microservice prototype DataOutputClient of Data Handler service

```
package aa8409.thesis.prototype.datahandler.clients;

import java.net.URI;
import org.apache.commons.lang.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpStatusCodeException;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;
import aa8409.thesis.prototype.datahandler.models.HttpException;
import aa8409.thesis.prototype.datahandler.models.NormalizedData;
/** Client for data output system access. */
@Component
public class DataOutputClient {
    /** Key of the environment property containing URL of the output service. */
    private static final String OUTPUT_URL_PROPERTY
        = "aa8409.thesis.prototype.datahandler.outputServiceUrl";
    /**
     * Key of the environment property containing authentication
     * token of the output service.
     */
    private static final String OUTPUT_TOKEN_PROPERTY
        = "aa8409.thesis.prototype.datahandler.outputServiceToken";
    /** The current environment. */
    private final Environment environment;
    /** The current rest template.*/
    private final RestTemplate restTemplate;
    /**
     * Create a new service.
     * @param environment  The current environment.
     * @param restTemplate The current rest template.
     */
    public DataOutputClient(@Autowired Environment environment,
            @Autowired RestTemplate restTemplate) {
        this.environment = environment;
        this.restTemplate = restTemplate;
    }
```

```
    /**
     * Send normalized data to the output service.
     *
     * @param normalizedData Data to send.
     */
    public void send(NormalizedData normalizedData) {
        String uriStr = this.environment.getProperty(OUTPUT_URL_PROPERTY);
        if (StringUtils.isBlank(uriStr)) {
            throw new IllegalStateException(
                    "Service is missing required configuration: " +
                      OUTPUT_URL_PROPERTY);
        }
        URI uri = UriComponentsBuilder.fromHttpUrl(uriStr).build().toUri();

        // Token is optional, supporting both authenticated
        //  and unauthenticated versions.
        String token = this.environment.getProperty(OUTPUT_TOKEN_PROPERTY);
        HttpHeaders headers = new HttpHeaders();
        if (StringUtils.isNotBlank(token)) {
            headers.setBearerAuth(token);
        }
        HttpEntity<NormalizedData> http
           = new HttpEntity<>(normalizedData, headers);
        try {
            ResponseEntity<Void> response = this.restTemplate.exchange(
                    uri, HttpMethod.POST, http, Void.class);
            if (response.getStatusCode() != HttpStatus.OK) {
                throw new HttpException(
                        "Unexpected status from normalized data sending.",
                        response.getStatusCode());
            }
        } catch (HttpStatusCodeException httpException) {
            throw new HttpException(
                    "Failed to send normalized data.",
                    httpException.getStatusCode());
        }
    }
}
```

Appendix 11. Cloud prototype lambda function of gitlab_json_source

```
# Copyright (c) AA8409@student.jamk.fi 2022.
# This file is part of Thesis Prototype 3.
from gitlab_client import fetch_data
import json
from os import environ
def lambda_handler(event:dict, context):
    """ Entry point of the Lambda function. """
    try:
        target = event['target']
        data = fetch_data(target, environ)
    except Exception as exc:
        raise Exception("Failed to fetch data from Gitlab Json Source!")
    return {
        "statusCode": 200,
        "body": json.dumps(data)
    }
```

Appendix 12. Cloud prototype gitlab client of gitlab_json_source

```
# Copyright (c) AA8409@student.jamk.fi 2022.
# This file is part of Thesis Prototype 3.
from datetime import datetime
from normalizer import normalize
from urllib.request import Request, urlopen
import contextlib
import json
def _get_and_validate(environment:dict, key:str, errors:list):
    """
    Read a single key from given environment and validate its value.
    Adds error message to given "errors" collection if invalid.
    Params:
        environment: The environment dictionary to read from.
        key: Key in given environment to read.
        errors: Error message will be inserted here if not valid.
    Returns:
        Value from given environment, or empty string.
    """
    result = environment.get(key, "")
    if not result or not result.strip():
        errors.append("Missing required environment variable: " + key)
    return result
```

```
def fetch_data(target:str, environment:dict):
    """
    Fetch data from GitLab JSON source.

    Params:
        target: Defines the data file to fetch.
        environment: The environment dictionary to read from.

    Returns:
        Data in normalized format.
    """

    # Building the request.
    # Parsing values from the current environment.
    validation = []
    url= _get_and_validate(environment, "GitLab_BaseUrl", validation)
    project= _get_and_validate(environment, "GitLab_Project", validation)
    folder= _get_and_validate(environment, "GitLab_Folder", validation)
    branch= _get_and_validate(environment, "GitLab_Branch", validation)
    token= _get_and_validate(environment, "GitLab_Token", validation)

    if validation:
        print(f"Environment is not valid: {validation}")
        raise Exception({"message": validation})

    full_url = f"{url}/api/v4/projects/{project}/repository/files/{folder}{tar-
get}.json/raw?ref={branch}"
    request = Request(url=full_url)
    request.add_header(key="Authorization", val=f"Bearer {token}")

    # Using "contextlib" to handle automatic disposing of the request.
    data = None
    with contextlib.closing(urlopen(request)) as response:
        data = response.read()

    if not data:
        print("Received data is empty.")
        raise Exception("No data!")

    data = json.loads(data.decode("utf8"))
    result = list(map(normalize, iter(data)))
    return result
```

Appendix 13. Cloud prototype lambda function of prototype_3_output

```
# Copyright (c) AA8409@student.jamk.fi 2022.
# This file is part of Thesis Prototype 3.

from output_client import send_data
import json
from os import environ

def lambda_handler(event, context):
    """
    Entry point for the AWS Lambda function.
    """
    try:
        data = json.loads(event["responsePayload"]["body"])
        send_data(data, environ)
    except Exception as exc:
        raise Exception("Prototype 3 output failed.")

    return {
        'statusCode': 200,
        'body': json.dumps("Prototype 3 output was success.")
    }
```

Appendix 14. Cloud prototype output client of prototype_3_output

```
# Copyright (c) AA8409@student.jamk.fi 2022.
# This file is part of Thesis Prototype 3.

from dataclasses import asdict
from urllib.request import Request, urlopen
import contextlib
import json

def send_data(data:list, environment:dict):
    """
    Send data to the output system.
    """
    url = environment.get("Output_Url","")
    token = environment.get("Output_Token","")

    for index, row in enumerate(data):
        _send_data_row(row, url, token)

def _send_data_row(dataRow: dict, url, token):
    """
    Send a single data row to given url.
    """
    body = _data_to_json_string(dataRow).encode("utf-8")
    request = Request(url=url, method="POST", data=body)
    request.add_header("Content-Type", "application/json; charset=utf-8")
    request.add_header("Content-Length", len(body))
    if token:
        request.add_header(key="Authorization", val=f"Bearer {token}")

    # Using "contextlib" to handle automatic disposing of the request.
    code = None
    with contextlib.closing(urlopen(request)) as response:
        code = response.getcode()

    if code != 200:
        raise Exception(f"Unexpected result: {code}!")

def _data_to_json_string(data: dict):
    """
    Convert given data to json string.
    """
    return json.dumps(data)
```