universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„FactCheck++ Reference Architecture based on Azure Cloud Services“

verfasst von / submitted by

Roman Szabo, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

Foremost, I would like to thank my supervisor Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas for his guidance, help and constructive feedback.

Further, I would like to express my thanks to Dipl.-Ing. Dr. Peter Kalchgruber for helping me understand the whole FactCheck++ framework, and providing valuable comments and feedback for the many architecture stages that were presented.

A big "Thank You" goes also to the fellow FactCheck++ students who participated in the workshops. Especially, I would like to thank Marie Aichinger and Elisabeth Wyslych for the valuable discussions which helped me understand the needs of the individual components well and well as helped to shape the final result.

Finally, I would like to thank my whole family for support and encouragement during my whole studies.

Abstract

Today's web is an open ecosystem comprising data to be accessed by everyone without any centralized content control. Therefore, much information on the internet is misleading, not verified, or untruthful, and they are published intentionally or unintentionally every day. The average user is not an expert in every field and cannot always make a correct decision on which internet sources are based on facts and verified information. To help mitigate this problem, a FactCheck++ project was proposed. Currently, the project is at the development stage and needs a reference architecture. This need is triggered by efforts to move towards a public cloud environment. The cloud-based reference architecture unites the individual implementation efforts, bring clear structure, and define optimal cloud services. Following the need, the system and reference architecture for the FactCheck++ based in Azure is proposed. Furthermore, an Azure setup proposal is provided with an implemented set of templates for creating Azure resources. The proposed architecture shows high potential when evaluated and proves to impact the ongoing and future research projects under the FactCheck++ project.

Kurzfassung

Das heutige Internet ist ein offenes Ökosystem, das Daten umfasst, auf die jeder zugreifen kann, ohne dass eine zentrale Inhaltskontrolle stattfindet. Daher sind viele Informationen im Internet irreführend, nicht überprüft oder unwahr, und sie werden jeden Tag absichtlich oder unabsichtlich veröffentlicht. Der Durchschnittsnutzer ist nicht in jedem Bereich ein Experte und kann nicht immer richtig entscheiden, welche Internetquellen auf Fakten und geprüften Informationen beruhen. Um dieses Problem zu entschärfen, wurde das Projekt FactCheck++ vorgeschlagen. Derzeit befindet sich das Projekt in der Entwicklungsphase und benötigt eine Referenzarchitektur. Dieser Bedarf wird durch die Bestrebungen ausgelöst, zu einer öffentlichen Cloud-Umgebung überzugehen. Die Cloud-basierte Referenzarchitektur sollte die einzelnen Implementierungsbemühungen vereinen, eine klare Struktur bieten und optimale Cloud-Dienste definieren. Ausgehend von diesem Bedarf werden das System und die Referenzarchitektur für FactCheck++ auf Azure vorgeschlagen. Darüber hinaus wird ein Azure-Setup-Vorschlag mit einem implementierten Satz von Vorlagen für die Erstellung von Azure-Ressourcen bereitgestellt. Die vorgeschlagene Architektur zeigt bei der Bewertung ein hohes Potenzial und beweist, dass sie sich auf die laufenden und zukünftigen Forschungsprojekte im Rahmen des FactCheck++-Projekts auswirken wird.

Contents

List of Figures

Listings

1. Introduction

In software development, architecture plays a crucial role in defining the product's technological landscape, expansion possibilities, and final quality from the early days. Architecture design is a prerequisite for any development work as it provides a clear structure of the system, tools to use, and design principles for the software. The resulting architecture is based on design, technical, and expansion requirements. The reference architecture is not strictly tied to a single software product, as it embodies accepted industry best practices and suggests the optimal services for specific deployment scenarios [12]. The anticipated scope of the reference architecture in the scenario for the research is to provide a tailor-made reference architecture composed of a system architecture proposal and a set of best practices with a reference for optimal services.

Today, a cloud computing concept is a well-established concept in computer science, on which a whole industry is built. The cloud industry provides access to state-of-the-art hardware and services at no upfront cost. These possibilities were brought by the big cloud providers such as Amazon Web Services[1], Microsoft Azure[2], or Google Cloud[3]. These providers provide a public cloud, which means that they are providing access to their services, which the customer can run for his product. Since the hardware is shared between many users, maintenance costs are shared, too, creating a compelling price structure for an individual. Easy access and fair pricing create an ideal environment for start-up software projects from the cloud.

Today's web is an open ecosystem comprising data to be accessed by everyone without any centralized content control. Much information on the internet is misleading, not verified, or untruthful, and they are published intentionally or unintentionally every day. The average user is not an expert in every field and cannot always make a correct decision on which internet sources are based on facts and verified information. To help tackle this challenge, the FactCheck project was born. The first vision [17] of the FactCheck project concentrates on detecting conflicts in the structured data[4] embedded in the web pages and presenting the conflicts to the users through a browser extension. This core concept was later expanded by exploring the conflict detection possibilities in unstructured data[5], multimedia content such as images, and forming a whole software product under the name FactCheck++. The FactCheck++ project, running under the supervision of the Multimedia Information Systems research group at the Faculty of Computer Science of the University of Vienna, consists of multiple student research projects not only on

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/
[3] https://cloud.google.com/
[4] data encoded in standardized structure in machine-readable format (i.e., JSON-LD, RDFa)
[5] data without standardized structure in human-readable format (i.e., written text)

fact-checking fronts but also in the fields of the user experience, user handling, feedback services, and many more.

Such software product design has a clear technical need, a system architecture. As the product is still in development, and the concept allows for future system expansion, a reference architecture is better suited in this scenario. Given the compelling offer of the cloud ecosystem, the potential of using various cloud services, and the learning potential for the students, an idea of designing the architecture for the cloud was born. Thus, the three main topics united created the base for this research - proposing a reference architecture based in the cloud for the FactCheck++ project.

1.1. Motivation

As briefly outlined in the previous section, there is a need to have a reference architecture for the FactCheck++ system. The main reason is to align the implementation results of the research projects. As mentioned previously, there are multiple ongoing research projects on various topics, where amongst expected results is an implementation of specific functionality for the FactCheck++. In the end, these results have to be able to be deployed and function together and be extended and maintained in the future. Therefore, to ensure the system's compatibility, using the same technology stack, following design principles, and keeping the system performant, a reference architecture containing all this information is needed. The implementation efforts are steered by the architecture, as each implementation effort needs to comply with it.

The effort to use the cloud environment as the deployment target motivates this research in the direction of cloud-native applications. However, such motivation changes the scope, as the overall design approach of cloud architecture varies from the standard on-premise scenario. Such a cloud-based architecture should follow sound principles of cloud computing technology. Thus, confirming the system's suitability for the cloud environment is in the scope.

During the initial analysis was determined that in the scientific literature, the specification and a cloud architecture design for a project like FactCheck++ is missing. The approach of the FactCheck++ project is asking for some specific requirements, and it is of high interest to find out whether one can come up with a suitable cloud-based architecture to meet those needs. Hence, this thesis addresses the topic of a general, suitable cloud-based architecture for similar systems (which is missing in the literature).

1.2. Research Questions

This thesis evolves around the main research question: "Is it possible to create a feasible cloud architecture for the FactCheck++ project?" This question can be directly followed by a similar sounding question: "Can be FactCheck++ a cloud-native[6] application?". Although both questions have a similar meaning at first glance, they question something

[6]application designed solely for the cloud, leveraging from all the cloud possibilities

different. The first part of the first question asks if we can create a cloud architecture for the FactCheck++, which is possible with many possibilities, in the result set S. The second part limits the result set S since not all possibilities follow sound cloud design principles, are performant, and are cost-optimized. The second question, asking if the application can be designed as cloud-native, impacts the final proposed architecture, as not all applications in the cloud are truly cloud-native.

To further shape the resulting architecture, an initial set of requirements were set:

- Provide optimal services for individual components
- Provide interface design between the components
- Scalable architecture by design (easy out-of-the-box scalability)
- Reasonably performant
- Cost optimized
- Provide monitoring and telemetry options
- Easily extendable architecture design

1.3. Research Approach

The research is approached in four main phases: (i) researching the FactCheck++ component design, current implementations, and collecting any known requirements from the individual projects, (ii) researching Microsoft Azure cloud provider, provided services, best practices, (iii) creating the final proposal by selecting the optimal services for the individual components, and (iv) prototypical implementation.

The research of the cloud providers is omitted, as the cloud provider of choice is Microsoft Azure. As cloud provider is defined for this research, it focuses only on services provided by Microsoft Azure.

The FactCheck++ research collects information on the current system status and implementation. From this information, it can be determined if the application can be designed as cloud-native. This information limits the possible architecture and thus limits the further investigation of the cloud services.

The cloud research involves researching the cloud architecture design patterns and recommendations, followed by Microsoft Azure research. Here, the recommendations from the Azure are collected, and a set of possible services is defined.

Finally, the architecture is designed in design iterations, and optimal services are selected. The iteration consists of a designing phase and a feedback phase, wherein the feedback is considered to improve the design in each iteration.

Once the proposal is created, a prototypical implementation is produced. This prototype consists of an example environment and a set of templates for creating this environment. The resulting proposal is evaluated on this prototype as well.

1.4. Outline

The remainder of the thesis is organized as follows. Chapter 2 provides background knowledge for a better understanding of the topics discussed in the next chapters. The chapter 3 provides literature background of this thesis. Chapter 4 discusses the design approach taken and provides the proposed architecture, followed by chapter 5 which fills the proposed architecture with actual Azure setup and provides the details about the implementation of the templates. Chapter 6 evaluates the results. Chapter 7 concludes the thesis with a summary and outlook for the future work.

2. Background

This chapter presents the background knowledge for a better understanding of the topics. It covers mainly the description and current state of the FactCheck++ project and Microsoft Azure and Azure services.

2.1. FactCheck++ Project

FactCheck++ is a Multimedia Information Systems research group project aiming at fact recognition on the web. The current scope of the FactCheck++ framework, as defined in [17], is to detect and resolve conflicts in the structured data and confirm correctness. "The framework offers three benefits: (i) it initiates and assists to solve conflicts directly at the data source, (ii) does not require complex data fusion algorithms, and (iii) helps improve the consistency of the web, implicitly allowing applications to be built upon more accurate data. [15]"

Structured data encodes data in one commonly used format (such as JSON-LD, Microdata, RDFa) using pre-defined schemas (e.g., schema.org). These schemas are developed by Google, Bing, and Yahoo mainly for user-friendly content display (e.g., biography data directly displayed in Google search results or Question and Answer tabs). These structured data are commonly present on web pages and are easily readable for computers. However, despite the overall availability of the structured data, it is often conflicting on different web pages, leading to false information presented to the user.

The FactCheck++ framework proposes a solution by detecting such conflicting data and proposing resolutions for them. The detection is done by comparing the entity representation detected from the visited website to the real-world entity. The entity is a condensed representation of a real-life object encoded in a pre-defined schema (e.g., Donald Trump as a person or Ghostbusters as a movie). The real-life entities are collected from multiple trusted sources, stored as a knowledge base, and considered facts. The instances (entity representations detected from the visited web page) are compared to the facts in the knowledge base. The two entities are compared based on precision metrics. A precision metric defines how much can two entities deviate to be still considered equal. Such fuzzy comparison is needed, as information is often not encoded in the same format across different web pages (e.g., publish date for a movie is on the first page provided as date-time format, on the other page, just month and year is provided). Finally, the comparison results for the visited web page are collected and presented to the user. This report consists of

1. entities found and considered as facts,

2. entities found and considered as conflicting,

3. entities found but missing in the knowledge base

4. entities not found on the visited page

As computer-based conflict resolution is only as good as the data and precision metrics provided, feedback is collected and analyzed to improve metrics used and application. The feedback is collected from the standard users and professional reviewers.

A dashboard web application is part of the project for data analysis purposes and system monitoring.

The previous paragraphs were inspired by the excellent summary of the FactCheck++ project in [15].

2.1.1. Current Architecture of FactCheck++ Project

As introduced in the vision paper [17], the initial FactCheck framework provided an initial architecture diagram, which was later revised and updated in [18] to represent the current state of the FactCheck++ project. The current diagram, as shown in Figure 2.1 is a simplified diagram only providing the proposed services and their interaction points, without providing runtime environment information.

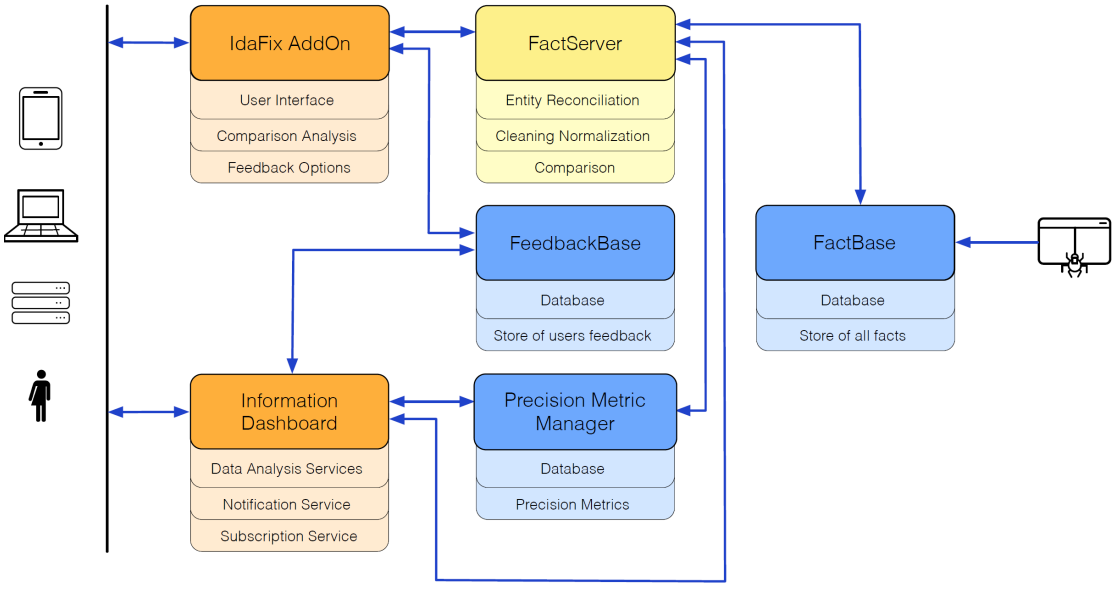


Figure 2.1.: Original FactCheck++ Architecture Diagram [18]

The following component descriptions are based on the component description presented in [17] and recently summarized in [15].

FactBase or knowledge base is a central storage component. It stores all the collected instances of the entities (our facts). The FactBase can be populated and maintained by one of these three methods: (i) adding an already existing collection, (ii) using web crawlers to collect data on the internet, and (iii) client software such as IdaFix AddOn. The FactServer primarily uses the FactBase for the comparison process.

FeedbackBase is a collection point of all user feedback provided for the comparisons made by the system or to the system self (e.g., a bug report). The information collected is categorized and analyzed to further improve precision metrics used and user experience.

Precision Metric Manager The Precision Metric Manager component stores precision metrics and related data and provides endpoints for requesting the used metrics and manual management. Additionally, the component internally uses unsupervised learning to adjust existing or create new metrics based on the analyzed feedback from the FeedbackBase.

FactServer is a central computing service consisting of a set of well-defined APIs. These APIs provide the following functionality: (i) retrieving instance data from the FactBase, (ii) comparison of instances. The FactServer also collects statistical data and updates or populates the FactBase with comparison requests.

IdaFix AddOn and Information Dashboard are user interface components. The IdaFix AddOn collects structured data from the web page, sends comparison requests, and displays the results. Users can also provide feedback through the AddOn. The Information Dashboard is provided to users, such as data analytics, researchers, and content providers. This dashboard utilizes various information about the system, easily consumed by target users. An example of such information is an analytics dashboard of conflicts or received feedback.

2.1.2. Current Runtime Environment

The current FactCheck++ prototype is deployed to some dedicated servers provided by the University of Vienna.

The IdaFix AddOn is a browser extension, and therefore the runtime environment is the user's browser. Prototype is implemented as Chrome[1] extension.

The Information Dashboard is deployed on a web server provided by the University of Vienna.

The FactServer prototype is implemented in Python[2] and deployed as a Rest API server. It provides a set of well-defined APIs for data retrieval, and executing comparisons. IdaFix AddOn and Information Dashboard use these APIs.

From the data store back end resources (FactBase, FeedbackBase, and Precision Metrics Manager) is only the FactBase implemented. It is implemented as a single instance of the

[1] web browser, https://www.google.com/chrome/

[2] programming language, https://www.python.org/

Apache CouchDB[3] NoSQL database. The CouchDB is deployed at a server provided by the University of Vienna.

2.2. Reference Architecture

A reference architecture provides recommended structures and integrations of IT products and services to form a solution. The reference architecture embodies accepted industry best practices, typically suggesting the optimal delivery method for specific technologies[12]. As per the definition of reference architecture, the benefits are that the FactCheck++ project will have tailor-made reference architecture. The main advantages of this approach are directly named in definition, mainly following best practices of software design, using optimal services, and meeting predefined performance criteria. In addition, such document anticipates — and answers — the most common questions that arise and avoid delays and further effort that may occur without using a tested set of best practices and solution approaches[12].

2.3. Cloud Native

"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."[19] This is the official definition of the Cloud Native Computing Foundation[4]. In other words, the cloud native systems take full advantage of the cloud service model, meaning that they extensively use Platform as a Service model [35]. The difference between the cloud and on-premise is the use of different service model. Whereas the on-premise data centers usually use a Pet service model (more resources on single machine, one server not working everyone notices), the cloud uses a Cattle service model (single use virtual machine, one server down, nobody notices) [35].

2.4. Microsoft Azure

Microsoft Azure is a public cloud offering from Microsoft, with various Microsoft or third-party products provided on the platform. Due to the vast library, this research will focus on Microsoft-provided services. These services are well documented, include license if needed, support is provided (except for "Preview" services), and have SLAs (Service-Level Agreements) covered by Microsoft directly.

The Azure services are categorized into categories such as storage, compute, monitoring. Still, the main distinction is between Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) offerings. This distinction is mainly in the computing section, where

[3] https://couchdb.apache.org/
[4] https://www.cncf.io/

both offerings are available based on how much control over individual instances is needed [24][25].

Azure has many benefits, but the main ones for this project are the offer of services, conductibility, usage, and the ease of use for beginners. The offer contains many services ready to use, meaning that almost all services require none or minimal configuration before use. The use of Azure services is straightforward and well-documented. Also, the services provide many integration options with different systems or deployment pipelines.

As always, there are also downsides. This includes the limitation of the services, meaning that where on the one hand are many options of services and configurations, on the other hand, some constraints define what can be deployed to the services. Another downside is less control over the individual instances, mainly in the PaaS setup. To sum it up, one can observe that the downsides mainly oppose the benefits, and the key to successful use of the offerings, in general, is finding the right balance.

2.5. Azure Services

Since the central part of the thesis covers the Azure architecture of the project, it is anticipated that the reader has general knowledge about Azure and Azure Resources used. To ensure understanding and reasoning, the Azure services used will be introduced and described. This section is highly dependent on the Microsoft Azure Documentation[5] and the Azure and Azure documentation version. Since the environment is constantly under development and changes are being introduced frequently, the information presented in this thesis considers the current state of the Azure. This information might change over time.

The description of the services will follow a following structure: (i) a brief description of the service is provided, (ii) pros and cons in the FactCheck++ context, (iii) pricing model, and (iv) development remarks – how "developer friendly" is the resource.

2.5.1. Resource Group

Description Resource Group is a logical container in which are individual resources organized. For easier representation, it can be viewed as a folder for the resources. Each resource deployed has to be in one Resource Group. The limitation of the Resource Group is that it cannot be nested (i.e., a Resource Group cannot be within another Resource Group). [33]

Pros

- Organization of resources in "folders"
- Access can be defined for whole resource group

[5] https://docs.microsoft.com/en-us/azure/?product=featured

Cons

- Missing nesting functionality
- Too vague definition bringing more questions than answers for correct usage

Development Remarks As Microsoft does not clearly define resource Group usage, some best practices treat the Resource Group as a logical container or an access scope. With Azure Role-Based Access Control (RBAC), access to an individual Resource Group can be limited. Furthermore, a naming convention should be defined within the project to keep the resource groups organized.

2.5.2. Azure Virtual Machines

Description Azure virtual machines is an Infrastructure as a Service compute offering from Azure. The virtual machine (VM) offering provides users with the operating system, storage, and networking capacities to run a wide range of applications. The VMs are needed in scenarios where there is a need for more significant control over the environment than App Service or other managed cloud services offer. To create the virtual machine, the user needs to select the preferred or needed operating system and size mainly. Azure provides the operating system images for Windows also with licenses and major Linux images. The sizing of VMs is divided by use (General Purpose, Compute Optimized, Memory Optimized, Storage Optimized, GPU, High-Performance Compute). For each category, there are multiple sizes with different specifications. [23]

Pros

- Full control over the settings, operating system, and software
- Backup and restore options
- Scalability options
- Pay as you go model or discounted yearly prices

Cons

- Higher operation overhead as managed services
- Paying also for an idle machine

Pricing Structure The VMs are charged hourly, meaning that every hour the machine is running, the subscription is charged. Azure hourly rate depends on the pricing model, operating system, and size. There are four pricing models – Pay as you go, one year reserved, three years reserved, and spot.

The pay as you go is the standard hourly pricing. The reserved models have a cheaper hourly rate, but total reserved capacity needs to be paid, even if not used.

The spot virtual machines allow taking advantage of unused Azure capacity at significant cost savings. However, when Azure needs the capacity back, the Azure infrastructure will kill and evict the Azure Spot Virtual Machine. Therefore, the Azure Spot Virtual Machines are great for workloads that can handle interruptions such as batch processing jobs, dev/test environments, large compute workloads [16].

The actual prices, pricing model details can be found on Virtual Machines Pricing page[6].

Development Remarks As the virtual machine behaves like a standard server or computer, the operation is no different from a standard computer. The developers need to set up the machine and then maintain it, since Azure is only responsible for providing running hardware, not its maintenance activities as operating system updates or security patching. Also, the networking setup (ports, any VNets, etc.) is managed by the developers or operation team.

2.5.3. App Service

Description Azure App Service is a PaaS offering for hosting HTTP-based applications like REST APIs, web applications, mobile back ends. The App Service supports either code deployment or container. The code deployment offers many languages or application servers to choose from, fully managed by Azure. The code can be in .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Applications run and scale with ease on both Windows and Linux-based environments. Baked into the app service is the ability to scale up/down or scale-in/out. Depending on the web app's usage, scaling the application up/down is done by increasing/decreasing the resources of the underlying host machine. These server resources are the number of cores, the amount of RAM available, or others. On the other hand, scaling out means increasing the number of machine instances. [23]

The computing and pricing depend on App Service Plan[7] (ASP). ASP is the computing resource required by App Service, and it is eventually where the application is running. It is analogous to conventional server farm hosting. The ASP defines the pricing tier, performance (CPU and memory), and scale rules of its applications. There can be multiple applications (App Services) running on one ASP. [3]

Pros

- No infrastructure maintenance efforts as it is managed by Azure
- Developer only need to provide the code
- built-in CI/CD support

[6] https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/overview
[7] https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans

- Easy to create scale up/down rules

Cons

- Paying also for an idle or stopped application

Pricing Structure The price depends on ASP, meaning that the user is not charged for the App Service itself but for the ASP only. The ASP pricing is based on tiers, individual scale settings, and selected OS. There are three primary groups: free and shared, dedicated and isolated. The fourth type is Consumption, but it is only available for the Function App.

The shared and basic tiers used shared resources and have no scale-out ability. In addition, the shared tier is only available on Windows.

The dedicated tiers (Basic, Standard, Premium, and Premium V2) use dedicated VMs where the resources are shared only within apps in the ASP. The scale-out option is available. It spawns a new instance of the ASP, meaning that the scale-out affects all applications running within the ASP.

Development Remarks App Service is a very convenient service for deploying servers or web apps. The developer only needs to deploy the compiled code to the specified place (Manual FTP deployment) or use CI/CD pipeline. The ASP resource linked to App Service is usually configured once and does not require any maintenance or other work during the project's lifetime. It is only used if scale settings or size needs to be adjusted.

2.5.4. Static Web Apps

Description The static web app is a new addition to the Azure family providing cheap web hosting options. The static web app provides an option to serve static content. The way the content is served is not from one place as a standard web server, but rather the static content is globally distributed, bringing the data geographically closer to the users shortening the load time. [5]

The static web apps also provide integrated support for Azure functions, providing serverless API for smaller projects without the need for separate resources. Key points are seamless integration with GitHub or Azure DevOps, security integration with AAD and others, SSL certificates, custom domains, etc. [5]

Pros

- Easy to develop and deploy web applications
- Out of the box distribution of static content
- built-in CI/CD support

Cons

- No control over content distribution

Pricing Structure Pricing structure for Static Web Apps[8] is simple. There are two tiers, Free and Standard. The difference is price, SLA, and storage quantity included in tiers.

Development Remarks The Static Web Apps is a straightforward service providing the missing service for deploying web applications. It has a straightforward structure and deployment options with CI/CD integration. The service provides basic web hosting options with optional Azure Function API.

2.5.5. Azure Functions

Description Azure Functions is a serverless solution from Azure. Just code needed for the problem at hand needs to be provided, without worrying about a whole application or the infrastructure to run it [23].

Azure Functions aim to process data, system integration, building APIs, and microservices. The function is a code block, which implements the system logic, where each can run independently (e.g., one REST call). The trigger is responsible for starting the code execution, where there are many triggers like HTTP trigger (for REST calls), event trigger (e.g., Event Grid event), timer, and many more.[4]

There are three pricing plans for running/hosting the functions: consumption, premium, and dedicated. A dedicated plan is an App Service Plan, the same as described for Web Apps. Consumption and a premium plan dynamically add or remove Azure Functions hosts based on incoming events. The main differences are that consumption plan functions have a timeout, where the premium plan offers unlimited runtime, VNet, premium instance sizes, and different billing structure. Apart from the pricing plan, each Function App also requires a general Storage Account for operations such as managing triggers, logging executions, and others.[23]

Scaling of the Function App infrastructure is based on adding CPU and memory by creating additional host resources based on a number of events triggering the function. The consumption plan host is limited to 1.5 GB of memory and 1 CPU, where an instance of a function host is the entire function app, meaning that all functions within one function app resource share the same runtime environment and scale simultaneously [23].

Function Apps also provide Azure Active Directory and CI/CD integration. Languages available are C, Java, JavaScript, PowerShell, Python. There is also an option to deploy custom handler and options to use Linux or Windows OS. [4]

Pros

- Easy integration with other Azure Services

[8] https://azure.microsoft.com/en-us/pricing/details/app-service/static/

- Many triggers types to choose from
- Consumption plan
- Automatic scaling of the functions

Cons

- No or minimal control over the runtime environment
- Limited code complexity forced by timeout in the consumption plan

Pricing Structure Pricing is based on plans. The Dedicated plan (ASP) has the exact pricing as the ASP pricing described above. The Consumption-based plan is billed based on the number of executions and execution time, whereas the Premium plan is billed based on CPU time and Memory time used for execution. Apart from execution time, the required Storage Account is billed separately with the standard rate. The same applies to networking rates. More information and details can be found on the pricing page[9].

Development Remarks Developing for Azure Function is straightforward. The developer only writes the code for the logic, no need for any complex server setup or other configuration code, as Azure handles all infrastructure. Also, there are easy deployment possibilities, either CI/CD pipeline or by use of Function Core Tools[10].

2.5.6. API Management

Description API Management is a service for creating consistent, modern API gateways for back-end services. It helps to organize, publish and maintain APIs accessible internally or externally. API Management provides the core competencies to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. [42]

The API Management comprises API Gateway, Azure Portal, and Developer Portal. The API Gateway is the endpoint that accepts the requests, verifies credentials (API keys, JWT token, etc.), enforces quotas, transforms calls (based on setup), and forwards the call to the corresponding HTTP endpoint. The Azure Portal is for administration to define the API schema, package products, set up policies, manage users, and collect insights. The Developer Portal serves as a web interface for developers, where API documentation is published, try out option is provided, account and subscription management.[42]

Pros

- Option for the professional well-defined API

[9] https://azure.microsoft.com/en-us/pricing/details/functions/
[10] https://github.com/Azure/azure-functions-core-tools

- Build-in credential validation
- Options for setting up limit calls, transform body, and other rules
- Can be distributed around the globe
- Build-in option for account and subscription management for developers and third parties

Cons

- Complex service with three portals to manage, and many setup options
- Can be costly with large usage

Pricing Structure Since API Management is a complex service, the pricing structure[11] is complex. To simplify, it is based on tiers with a Consumption tier for pay-as-you-go shared service with limitations and five private tiers (Developer, Basic, Standard, Premium, Isolated). Each tier has a flat monthly fee for the service, where the higher tier provides bigger throughput, and the different number of scaling units is the main difference between tiers.

Development Remarks The development effort depends on the complexity and needs from the API Management. The setup effort is needed regardless of mapping the API calls, setting up authentication, and the developer portal. Other work depends on the number of rules or transformations required to be written. Overall, the complexity of developing the API Management is a bit higher due to multiple portals and custom XML structures used to write rules and transformations.

2.5.7. Azure Data Lake Storage Gen2 and Storage Account

Description Azure Blob Storage is a cloud storage solution. It is optimized for storing massive amounts of unstructured data. Unstructured data has no structure like text or binary data. The storage account is designed for storing documents, serving them directly to the browser, streaming audio/video, storing files for distributed access, writing logs, storing backup data or archiving, storing data for further analysis. [23]

Azure Data Lake Storage (ADLS) is built on top of Azure Blob Storage and provides file system semantics, file-level security, and scale. Next, the ADSL is optimized for use with analytics services from Azure with better performance thanks to a hierarchical namespace, which significantly improves performance on directory management operations and minimizes the need for copying data before analysis. The solution is also scalable as it can serve many exabytes of data with throughput measured in gigabits per second.[32]

[11] https://azure.microsoft.com/en-us/pricing/details/api-management/

Pros

- Optimized for high-performance data transfer
- Folder structure
- Cost-effective solution

Cons

- To keep cost-effectiveness, file tiers rules needs to be administered

Pricing Structure The pricing structure[12] is based on data amount and transactions. The cost per GB and transactions depends on the storage tier selected. The tier can be defined per file (Premium, Hot, Cool, Archive), where the price per GB is descending from Premium to Archive. Contrary, the transaction cost behaves oppositely - ascending from Premium to Archive. Example: The Hot tier has cheap access (cost per transaction) but a higher price per GB. The Archive tier has cheap storage but has expensive access (cost per transaction). Due to this fact, it is advised to set tier rules and do regular housekeeping of the files to optimize cost by assigning the correct tier to files.

Development Remarks Working with a storage account is pretty simple. There are a few ways to organize and upload files: (i) the REST API, (ii) Storage Explorer application, or (iii) container view from Azure Portal[13]. In addition, the storage account has interconnectivity to other Azure Services, in most cases through managed identity or connection string. There are also libraries for most languages to simplify the work with the Storage account.

2.5.8. Table Storage

Description Table Storage is part of the Storage Account service, providing a schema-less table solution. The Table storage can be used either as key-value storage or as a simple table store. Table storage can be viewed as a database table and row as an entity.[40]

Table Storage is the basic version of Table API. The Table API, part of the CosmosDB offering, is a premium table storage, providing secondary indexes, global distribution, and throughput optimized tables. The SDKs and structure is the same for both Table API and Table Storage.[37]

Pros

- Fast data look-up
- No schema
- Cheap storage and access costs

[12] https://azure.microsoft.com/en-us/pricing/details/storage/data-lake/
[13] still in public Preview

Cons

- Partition and Row keys need to be chosen wisely to keep the look-up performance

Pricing Structure Pricing structure[14] is very similar to the Blob Storage Account. The pricing is again based on the amount of data stored and the number of operations on the data. The cost per GB and per operation depends solely on availability zones, as there are no access tiers for Table Storage.

Development Remarks The work with table storage is simple and similar to relational databases. The differences are that there is no schema and fast look-up is only on partition key and row key column. This needs to be in mind, as these keys must be chosen wisely to index the tables efficiently. The benefit to using Table storage is that, if needed, it can be easily switched to a CosmosDB account without the need to modify the code since both use the same SDKs.

2.5.9. Azure Data Factory

Description Azure Data Factory (ADF) is a managed cloud service providing extract-transform-load (ETL), extract-load-transform (ELT), and data integration capabilities. The ADF is designed to process large quantities of data and perform transformation jobs or housekeeping jobs. The ADF is called Code-Free ETL, as the jobs are maintained from the developer portal, and no code is required for most transformations.[7]

The ADF is a tool providing data pipelines. It ingests the data, using its 90+ native connectors to connect to the data source, collect the data, and copy/store it in centralized cloud storage, e.g., Data Lake. When the data is available in the cloud, mapping, analytics, or transformation data flows can be run with the data. If the built-in transformation capabilities are not enough, a compute linked service as Hadoop, DataBricks, or others can be used.[7]

The top-level concepts of Data Factory evolve around pipelines. The pipeline is a job or logical grouping of activities in logical order. Activities are individual steps, e.g., connecting to the database or copying activity. Datasets represent the data structures in data stores used by activities as input or output. Linked services, as mentioned, are services that can be connected to ADF as activities (compute linked services) or as a data store (database, storage, etc.). Integration runtime provides the bridge between ADF and linked service. Various types of triggers can trigger the pipelines, have parameters for each run, and variables. All the pipeline runs are also monitored in ADF.[7]

Pros

- Able to process large data quantities
- Easy to develop and operate

[14] https://azure.microsoft.com/en-us/pricing/details/storage/tables/

- Great connectivity

Cons

- Might be too powerful/costly for initial development stages
- Requires knowledge prior proper use

Pricing Structure Pricing structure[15] for ADF is quite complex. The price is build-up from multiple sections such as pipeline orchestrations, execution time, and operations. The pipeline orchestration consists of run activities billed per run. Data movement, activity runtime, and linked service are billed hourly. The cost for this section differs if we run the activities in Azure, Azure VNet, or Self Hosted integration runtime.

Data Flow costs are execution clusters inside ADF used for data transformations at scale. The price depends on compute type selected - a general-purpose or memory-optimized computing service, and if a reserved plan is applied. The billing is per vCore-hour.

Lastly, the section of Data Factory Operations represents the Read/Write of entities (datasets, linked services, pipelines, integration runtimes, and triggers) and is billed per entity. In operation, the Monitoring is billed per retrieved monitoring records.

Development Remarks The ADF is a powerful service, but it requires knowledge to explore the possibilities fully. The development on the platform is simplified to a developer GUI, where all pipelines, integrations, transformations are implemented and monitored. The great advantage of the ADF is the ability to add linked services for data transformation/analysis.

2.6. Azure Resource Manager

Azure Resource Manager (ARM) is the deployment and management service for Azure. ARM provides a management layer for creating, updating, and deleting Azure resources and management features like access control, locks and tags to secure and organize resources after deployment. The ARM handles all requests from Azure tools (Azure Portal, Azure PowerShell, Azure CLI, Rest Clients).[33]

2.6.1. Azure Resource Manager Templates

Azure Resource Manager Templates (ARM Templates) is an Azure option for Infrastructure as a Code solution. The templates provide a means of automation of Azure resource deployments, minimizing the manual task/set-up of Azure resources, deploying a new environment, or redeploying existing or broken in a few simple steps.[30]

ARM Templates are JSON files defining the infrastructure resources and their configuration. In templates are specified resources and their configuration, the actual deployment execution is handled by Azure Resource Manager.[33]

[15] https://azure.microsoft.com/en-us/pricing/details/data-factory/data-pipeline/

Benefits of using ARM Templates are (i) simplified deployment and redeployment of the infrastructure, (ii) repeatable results, (iii) orchestration of resource deployment (in-order or parallel resource deployments), (iv) modular files (templates can be broken down to individual reusable components, linked at deployment time), (v) a test of templates, and (vi) adding bash or PowerShell scripts. The ARM template deployment can be integrated into CI/CD pipeline with tools like Azure DevOps to simplify infrastructure deployment further.[30]

As ARM Templates are part of the experimental implementation of the resulting architecture, a detailed introduction is presented in the following sections.

2.6.2. Template Structure

To fully understand the structure and contents of the ARM Template JSON file, a resource file for one resource will be broken down and explained. In the Listing 2.1 is depicted the default structure of ARM template.

```
{
    "$schema": "https://schema.management.azure.com/schemas
   /2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {},
    "functions": [],
    "variables": {},
    "resources": [],
    "outputs": {}
}
```

Listing 2.1: Default ARM Template Structure

Each template consists of parameters, functions, variables, resources, and outputs. The "parameter" section contains a declaration of each parameter used in the template. A parameter declaration is an object containing name, data type, default value, permitted value, and metadata. Next, the "functions" section provides the option to define custom functions used in the pipeline. Next, the "Variables" section serves the same purpose as "functions" but defines variables. Next, the "Resources" section contains all Azure resource declarations, which the template should deploy. Finally, the "Outputs" section contains a declaration of any output needed by subsequent template deployment or just as output value shown to the user.

2.6.3. Parameters

Deployment parameters values need to be provided at the time of deployment. Each value is validated against the declaration by ARM (Azure Resource Manager) in the validation step. In Listing 2.2 a sample parameter named "kind" is declared. The parameter name is the name of the JSON object. Inside the parameter object is always a key "type" with the parameter's data type. Supported data types[16] are string, secureString, int, bool,

[16]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/data-types

array, object, secureObject. Secure version for string and object can be used for handling secrets, passwords, keys, etc. These values are neither saved in deployment history nor logged. "Type" key is required. Other keys are optional. In the Listing 2.2 are optional keys used. The default value is used if no value is explicitly provided. Allowed values are defined as an array of all permissible values. For numeric value parameters, a range can be defined additionally. Lastly, the metadata object can pass a description of the parameter. This description is then displayed as an information tooltip in the Azure portal.

```
"parameters": {
    "kind": {
        "type": "string",
        "defaultValue": "StorageV2",
        "allowedValues": [ "StorageV2" ],
        "metadata": {
            "description": "Storage account kind/version"
        }
    }
}
```

Listing 2.2: Parameter Declaration

Depending on the deployment option used, the parameter value is provided differently. When deploying through the Azure portal, parameter values are directly provided in the portal. In the case of the Azure CLI or PowerShell deployment, the values are provided directly in the command. For all methods, the parameter file can be provided. A parameter file is a separate JSON file consisting of parameter names and values. An example can be seen in Listing 2.3. The parameter file is commonly used in automated deployments, multi-user, or multi-environment scenarios. In these scenarios, the resource section is the same, just parameters are changed (e.g., name, pricing tier). In these cases, each environment or user can have all necessary parameters customized in their parameter file, omitting the manual value input per deployment.

```
{
    "$schema": "https://schema.management.azure.com/schemas
    /2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "kind": {
            "value": "StorageV2"
        }
    }
}
```

Listing 2.3: Parameter Value Declaration

2.6.4. Resources

The resource section lists JSON objects describing all resources the template should deploy. The basic resource declaration requires an API version, name, location, type, and other resource-specific properties. The "type" is a resource type, e.g. `microsoft.insights/components` for Application Insights. API version determines which version of ARM REST API should be used. The API version and type determine additional resource-specific properties required or optional. Name is the user-defined resource name displayed in the Azure portal or used as part of the URL (e.g., for App Service). Usually, it adheres to a project-specific naming structure. The Listing 2.4 depicts an example of an Application Insights resource declaration. Common, tags, and resource-specific properties are used in this case. Here the resource-specific property is, for example, `properties` key, whereas value is an object consisting of other properties such as type of application or application ID. Apart from example resource declaration, the injection of parameter values into resource fields can also be observed. Each function call in the ARM template needs to be declared by using square brackets inside JSON value (`"[]"`). Here is used Build-in function `parameters(parameterName: string)` is used, which returns given or default value for the parameter.

```
{
    "type": "microsoft.insights/components",
    "apiVersion": "2015-05-01",
    "name": "[parameters('appInsightsName')]",
    "location": "[parameters('regionId')]",
    "tags": "[parameters('tagsArray')]",
    "dependsOn": [],
    "kind": "",
    "properties": {
        "ApplicationId": "[parameters('appInsightsName')]",
        "Application_Type": "web",
        "Flow_Type": "Bluefield",
        "Request_Source": "IbizaAIExtension"
    }
}
```

Listing 2.4: Resource Declaration

2.6.5. Deployment Options

Deployment of the ARM Template means submitting it to Azure Resource Manager. There are multiple possibilities, depending on which way suits the best in the given situation. The deployment options are:

- Command line tool (Azure Cloud CLI or PowerShell)[17]
- REST API[18]
- Azure Portal[19]

- "Deploy to Azure" button / GitHub[20]

Command Line Tool can be used to deploy the ARM templates to Azure. There is a possibility to use either Cloud Console (bash or PowerShell) versions in the Azure portal directly or locally installed Azure CLI or PowerShell. Both local and cloud versions function the same way, using a simple `az deployment group create` Azure CLI command or PowerShell counterpart `New-AzResourceGroupDeployment`. The resource group name, deployment name, and ARM template JSON file location are required command options. The parameter values can be supplied in the deploy command itself, as parameter files, or default value will be used. If no default is specified, the user will be asked to provide value in an interactive shell. An example of a deployment using Azure CLI to deploy the Main template with local parameter file is presented in Listing 2.5.

```
az deployment group create \
--name DeployMain \
--resource-group STUD_INF_Szabo_Roman_Thesis \
--template-file .\mainTemplate.json \
--parameters '@main.parameters.json'
```

Listing 2.5: Example of Azure CLI Deployment

REST API option provides HTTP endpoints and specification[21]. These endpoints can be used to deploy templates or query information about deployments. REST API is the master implementation where all other tools such as Azure Portal, SDKs, CLI tools translates commands into HTTP calls received and processed by the ARM to achieve consistent results [23].

Azure Portal deployment can be triggered either by deploying from Azure Marketplace[22]. A "Template Deployment" item needs to be searched in Azure Marketplace to deploy the ARM template [28]. This item allows the user to select the template from storage or upload it and deploy.

"Deploy to Azure" button is similar to the Azure portal deployment, where the template is loaded directly to the portal from the given URL. This method also referred to as the GitHub[23] method, is used to deploy templates from publicly accessible resources, in most cases GitHub. According to Microsoft Documentation, the "Deploy to Azure" button is a specific image with a link behind it. The link is composed of a standard part,

[20]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-cloud-shell?tabs=azure-cli
[20]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-rest
[20]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-portal
[20]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-to-azure-button
[21]https://docs.microsoft.com/en-us/rest/api/resources/deployments
[22]https://azuremarketplace.microsoft.com/en-us/marketplace/
[23]git-based version control, https://github.com

to which a URL encoded template URL needs to be appended. The standard URL part and any update from Microsoft can be found on the documentation page[24]. An example of button appearance can be seen in Figure 2.2 and Markdown (most used implementation) in Listing 2.6. In the Markdown snipped can be observed a link for the button graphic itself, common link, and URL encoded link to the template itself.

Figure 2.2.: "Deploy to Azure" Button*
* Source: https://aka.ms/deploytoazurebutton

```
[![Deploy to Azure](https://aka.ms/deploytoazurebutton)](https://portal.
    azure.com/#create/Microsoft.Template/uri/https%3A%2F%2Ffctemplates.
    blob.core.windows.net%2Ftemplates%2FARMtemplates%2FmainTemplate.json)
```

Listing 2.6: Markdown Example of the "Deploy to Azure" Button

2.7. Sound Principles of Designing Cloud Architecture

As can be seen by the descriptions of cloud architecture presented in the chapter, the cloud architecture varies from the traditional on-premise architecture. These differences are best observed by analyzing and performing a cloud migration project, where the current architecture and design principles are challenged by cloud principles. Such differences can be nicely observed in cloud migration guides such as Cloud Adoption Framework guidance[25] from Azure. Here are explained the migration guides and all the best practices.

As cloud services differ vastly from the standard datacenters, the architects and developers are forced to be more creative to deliver optimized services concerning costs and performance. Such design patterns are also taken in the FactCheck++ architecture, to provide the best architecture possible with the current state of the Azure offerings. It is not as simple as creating a microservice architecture for all scenarios. That is often a misconception that microservice architecture is the best in cloud environment and should be used for all applications [20]. That, in fact, is the opposite, the cloud journey should always start with the analysis of the application itself, but also analysis of the development team self. The team analysis is important to determine team skill set, which could shape the final decision of the architecture (e.g., an expert team of monolith server developers would have problems developing a microservice architecture in feasible time). The analysis of the application self determines the best cloud approach. It can be practically anything, with the cloud options, but usually the application falls into one of the standard categories, as monolith server, microservice application, on-demand processing (e.g., Azure Functions) and others. So, the chosen migration strategy (if just

[24]https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-to-azure-button
[25]https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/

rehost the application, refactor, rebuild, or reinvent) depends on the current state of the application, skill set of the team, and time and resources available for the job. Therefore, there is essentially not a right or wrong architecture, more like how close to the perfection it should be. This decision is ultimately left to the application owner, as he needs to balance out that three pillars (current state of the application, skill set of the team, and time and resources available). The outcome than shapes the outcome architecture.

To sum it up, the main architecture principle is to know the application. Each application is unique, and therefore the architecture has to be as well. The main difference is that cloud broadens up the horizon and allows to create architecture and infrastructure for the application, rather than designing application for the given infrastructure. This change brings naturally its own pitfalls. One of the major one is the missing skills in the development teams, skills that require time and money to get. Furthermore, having unique architecture for specific application or framework is great, this solution does not scale well in large enterprises with many applications. To overcome these issues, companies often leverage the cloud providers of choice in providing the trainings and expert help (e.g., Azure FastTrack[26]) to kick off and speed up the cloud journey. Enterprise tries to use standard architecture templates for most of the applications (e.g., simple web apps), but due to performance and cost optimization, they are partly forced to invent cloud solutions for certain applications. The sound cloud principles are to design cloud native applications, designed for PaaS services, are event-based or microservices. For big data solutions, the dedicated resources should be used. Containerized applications are welcomed as well. Nevertheless, the cloud offers many services, and the sound ones are those that fit best the needs of the application and team.

[26]https://azure.microsoft.com/en-us/programs/azure-fasttrack/overview

3. Related Work

This section will provide a brief overview of the fact checking and cloud architecture advantages to provide the literature background behind the thesis.

One may ask why not include technical architecture comparison from other fact checking projects. The reason for not presenting and discussing results from similar projects is that scientific literature is missing the specification or design of cloud architecture for similar projects.

3.1. Brief Overview of Fact Checking

The need for fact-checking efforts originates in the increasing use of the internet as an information source. With the increased popularity and lack of oversight, the internet, especially social media, is a breeding ground for misinformation. Although most false rumors are cleared after some time, the damage was already done. The spread of misleading information is increased by increasing the use of social bots[1] for economic or political gains. The widespread misinformation is also caused by a lack of fact checking solutions. The manual efforts are expensive, time-consuming, and not scalable. Automatic and semi-automatic fact checking tries to overcome these issues. However, even though these efforts are somewhat effective, social media are facing criticism for not taking strong enough measures based on outcomes of fact-checking.

There are three main categories of fact checking — manual, semi-automatic, and automatic. Manual detection is based on humans. Semi-automatic uses software to detect possible articles or information submitted for manual detection by humans, thus speeding the work by preselecting the texts of interest. Automatic checking addresses scalability issues and relies on language processing, machine learning, data and knowledge management, and graph theory. As the end users might not easily understand outcomes from (Semi)Automatic detection, various projects analyze and present the results, use external managed metrics and pressure on social media and search engines to include outcomes in their search results or posts.

This fact checking overview is a summary of [15].

3.2. Cloud Architecture

Cloud-based architecture style differs from on-premise computing style. The architectural style has been adopted to the cloud resources principles, as the cloud is a distributed

[1] computer algorithm that creates content and interacts with social media users

architecture of individual cloud-native services. It provides resources as services in a tiered fashion to construct a complete technology stack from hardware through middleware platforms to applications [**cloudArchitecture**]. The difference of on-premise computing is defining own hardware and runtime properties (operating system, software). On the other hand, the public cloud directly influences the architecture of applications by predefined runtime options and cost structure. The cloud-native[2] applications must cohere to cloud principles, and patterns [**cloudArchitecture**]. The significant change in application design is an emphasis on microservices, short-lived processes, and use of PaaS services [**cloudArchitecture**][41]. The reason for microservices and short-lived processes is cost optimization [41]. According to the cost comparison, the work of Villamizar et al.[41] shows that using the microservice approach can save approximately 15% compared to monolith servers and approximately 50% of costs with the use of AWS Lambda[3]. AWS Lambda is a comparable resource to Azure Function App.

3.3. Cloud vs. On-Premise computing for Businesses

The argument of which architecture is better, using cloud or on-premise, has been discussed since cloud launch. Despite all the benefits and negatives of both approaches, no one answer fits all businesses. It also depends on whether the business is migrating to the cloud environment or it is a startup like FactCheck++.

The choice of approach depends on an individual company, their current IT department technical knowledge, and budget. Businesses with not as confident IT departments tend to use cloud technologies more, letting the cloud provider take over the setup and support, with a transparent cost structure. On the other hand, with increasing business size, the IT departments tend to be more advanced, where the use of hybrid cloud[4] approach is often seen. This approach will better balance capital and operational expenses. Further details can be found in [13].

It is often easier and more cost-effective for startups and small businesses to use cloud offers, as the business can devote time to core business development [2]. The work of Carr[2] also provides a solution with all the crucial parts and decision trees for startup businesses. This work shows that pure cloud solutions are profitable, especially for small businesses.

For already existing businesses, not only with IT background, it might be interesting to consider the cloud with their SaaS (Software as a Service) products as described in [1]. The SaaS offers to enable even small businesses to use tools previously available only to big players. The services are delivered through web browsers, needing high-end hardware and a competitive pricing structure.

[2]application developed especially for cloud
[3]https://aws.amazon.com/lambda/
[4]combination of on-premise private cloud and public cloud

3.4. Azure Architecture

As Microsoft Azure wants users to deliver the best approaches, they offer, apart from documentation also a database of various system architectures, depicting the best use of their resources. Apart from existing architectures, also design principles of the cloud are provided. All of this is provided within Azure Architecture Center[5]. Although this collection of real architectures[31] is vast, it does not contain fact checking projects as such. Despite the lack of direct use case, this resource proves itself, as the design patterns presented in another use case scenario can be used in the FactCheck++ project self. One of the biggest influences of the storage design approach takes inspiration from "Medial data storage solution"[6]. In this case, the data is collected, analyzed, and provided to the user in analyzed and structural form. It forms a data pipeline with raw collected data as input and structured analyzed results as output. Another example can be found in "Loosely coupled quantum computing"[7]. Here, an API Management resource is used as a gateway to multiple Function Apps resources. Such design patterns can be later observed in the final architecture design.

To sum it up, Azure Architecture Center appears to be one of the best resources for Azure-based system architectures and showcases proper use of the components. Unfortunately, it is not a scientific research publication, despite it providing up-to-date and relevant information compared to mostly outdated Azure White Paper publications.

[5]https://docs.microsoft.com/en-us/azure/architecture/
[6]https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/medical-data-storage
[7]https://docs.microsoft.com/en-us/azure/architecture/example-scenario/quantum/loosely-coupled-quantum-computing-job

4. Design Approach

This chapter describes the taken design approach and design decisions. Firstly will be the overall architecture design discussed and afterward more details about the respective parts of the FactCheck++. Lastly, Azure Resource Manager templates implementation of the proposed architecture will be discussed concerning the proposed structure and design decisions.

4.1. General Overview

The FactCheck++ architecture is designed as a cloud-native application, meaning that it is designed to support scalability and leverage public cloud services [19].

Conceptual Design

The original architecture presented in Figure 2.1 was redesigned to a newer concept. Conceptually, the structure of the components is designed for the cloud. Function modules are adapted to the current stand.

The new conceptual diagram in Figure 4.1 represents proper split to front end, middleware, and back end.

Front End group did not change from original architecture. The IdaFix and Dashboard App are still present here. Functionality was not changed for these components.

Middleware group still consists of the main building block - FactServer. The difference is in the functionalities provided by the API. These were expanded with new API collections, already implemented, or implementation is in progress. Apart from adding the new functionality such as User management, the Precision Metrics Manager base was merged to FactServer. Because Precision Metrics Manager is more computational effort than data store, it makes more sense to include it as part of FactServer functionality and API set.

Back End group poses a more considerable change. The Bases are merged under a single XBase component, meaning that each Base is an individual database in a single component, opposed to the original design of multiple components. It simplifies maintenance of the system - single resource settings for all underlying bases (UserBase, FactBase, etc.). Moreover, connectivity and access control are simplified since this single component setup can define a single access point.

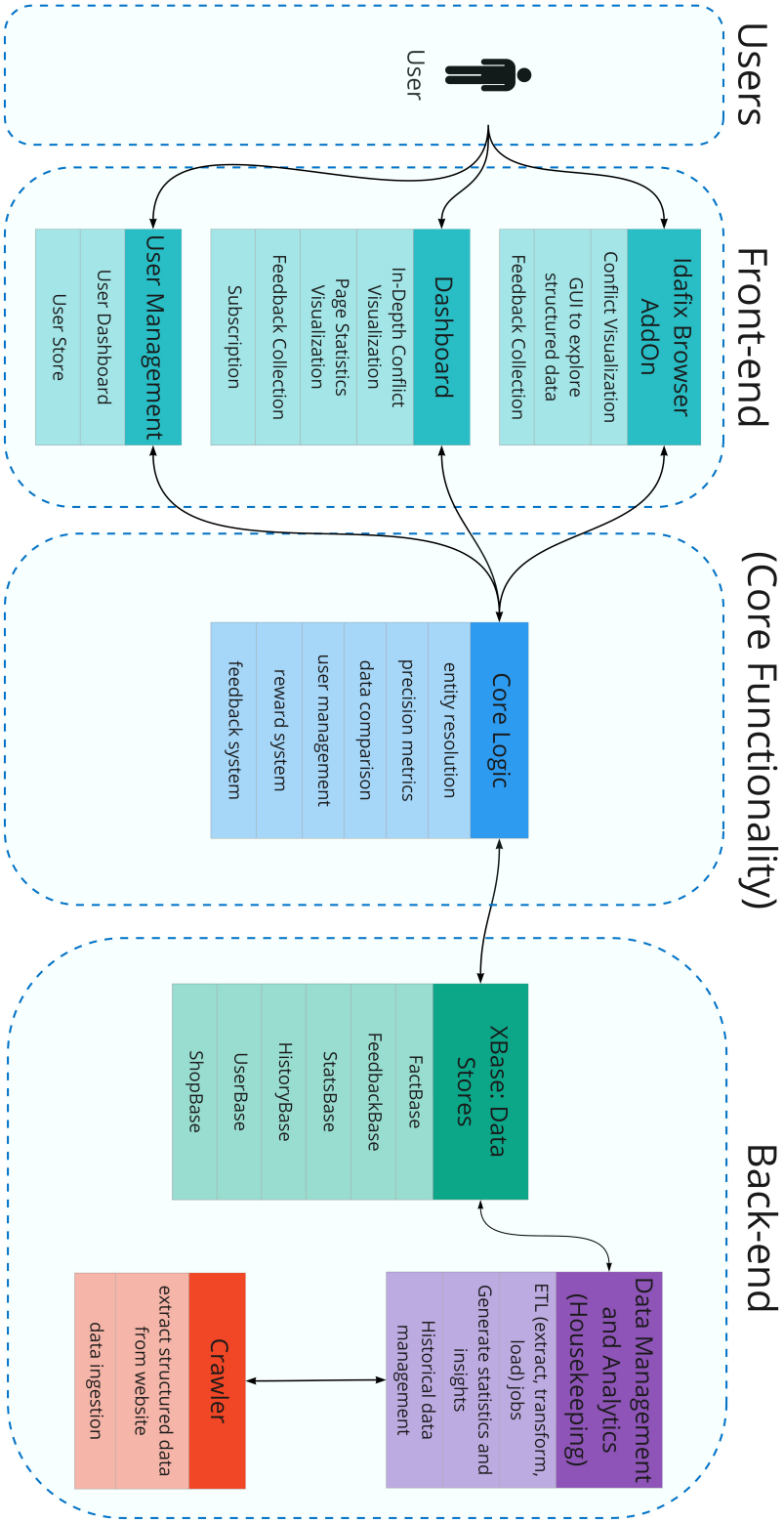


Figure 4.1.: Conceptual Overview of the Architecture

A new addition to the system is the Housekeeping component. This component is introduced with the new data storage approach proposed within the new architecture. The details are described in further sections, but shortly, this component is responsible for moving and transforming data within the system. Additionally, it provides a connection point for advanced data analysis.

The crawler component is just drawn as a regular system component rather than just as an icon.

Connections within the system are reconsidered and minimized. When looking at the original Figure 2.1, many connection links are present between resources, especially when front-end resources access data stores (FactBase, FeedbackBase) directly, resulting in no well-defined access and control flow. This behavior is changed in the new reference architecture concept. Due to maintainability and system security, it is needed to have defined connection points and interfaces between individual components of the system. Rather than having wild connections between components, in the new conceptual design, the FactServer serves as the "main gate" to the framework. This approach allows us to define that access to the XBase is only through FactServer, meaning that all access is controlled by FactServer (authentication and ensuring proper data visibility). Such an approach simplifies the overall setup and system maintenance, increasing the system's security.

Azure Design

The architecture consists of multiple Azure resources, split into resource groups. The separation of the resource groups is based on the logical separation of the application based on layers. The layers represent a logical section of the application (front end, back end, and data sources). They are used to group resources based on the layer where they belong. The groups are also used for the Role-Based Access Control (RBAC) concept for FactCheck++, which is in the scope of another parallel student project focusing on User Access and Rewards. RBAC allows administrators to control access to the individual resource groups based on the developer's needs. This approach follows the least privilege security principle [6] and is highly recommended by Azure since Azure Access control is built on RBAC principle [36].

Figure 4.2 depicts the general overview of the architecture. The figure uses icon elements provided from Microsoft[1] to represent the Azure services properly. The general overview description is split into sections ordered left to the right regarding the Figure 4.2.

4.1.1. Users

As presented in chapter 2, the FactCheck++ has two main user groups: standard and dashboard users. These users interact with the front-end services, represented by the IdaFix browser extension and Dashboard web application.

[1] https://docs.microsoft.com/en-us/azure/architecture/icons/

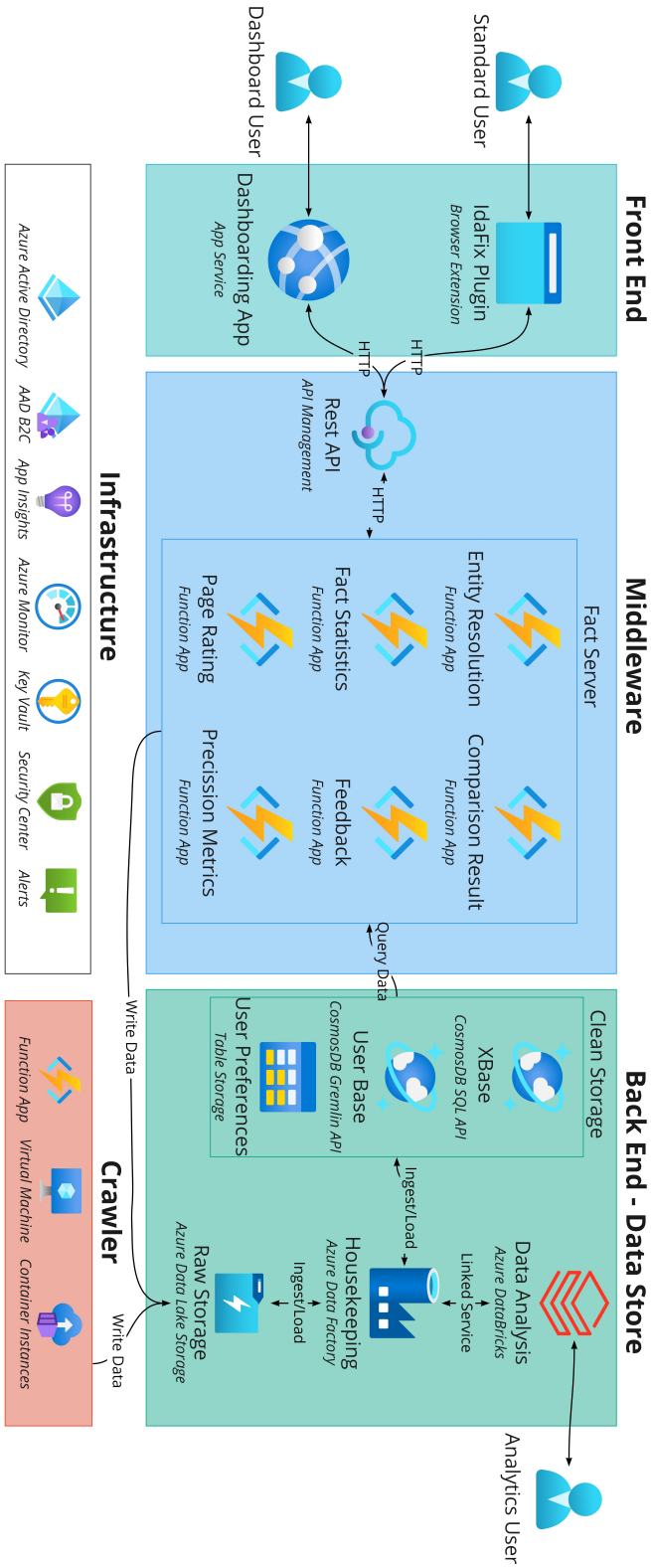


Figure 4.2.: Detailed General Overview of the Architecture

Apart from end-users accessing front-end services, there is also a particular type of user – Analytical user (data analyst). These users can access and run data analysis through an access point in the Azure DataBricks portal.

4.1.2. Front End Resource Group

The front end Resource Group (RG) is a collection of services needed for front-end applications. Since the IdaFix is an extension and does not need any Azure service, it has no dedicated Azure icon, but a browser icon is used. The extension is available to users from the browser extension store and runs within users' web browsers.

On the other hand, the Dashboard web application needs to be hosted on Azure. There are multiple options for hosting web applications on Azure, mainly the IaaS solution – a virtual machine – or one of the PaaS services like App Service, Static web apps, or container application. The PaaS solution is selected to satisfy the scalability and easy-to-use requirements. Therefore, the App Service Azure service is suggested for non-static web applications, as it provides an Azure managed environment, easily integrated with other services, and easy scalability rules setup. In case, at a later stage of the project will be found out that the application will be static (no running application on Azure, e.g., HTML Web Page) and only hosting will be required, the App Service should be substituted for the Static Web App offering, which is more optimized for static content hosting and more cost-optimized. The scalability requirement is satisfied as well. However, unlike the App Service, Static Web App distributes the static assets across the globe, meaning that the application contents are automatically distributed and physically close to all users, making the application serve faster [5], without the need to create multiple global replications manually. The costs for both (VM and App Service) are comparable and depend mainly on the computing power. In the case of Static Web App, there are currently two plans, free one and standard one.

To sum it up, the Dashboard app is designed to use PaaS solutions from Azure with either App Service or Static Web App for easy maintenance and leverage of Azure provided easy scalability setup.

4.1.3. FactServer Resource Group

FactServer RG or middleware services consist of multiple resources. When combined, they provide the functionality of the API server. At first glance, it might seem a bit over-engineered for an API server implementation. However, the effort was to leverage the cloud possibilities and provide a cloud-native serverless API that forms the FactServer. To explain why the serverless application is named FactServer, it is because logically, it is the Rest API server. However, since the application uses Azure Functions, it is considered a serverless application since it executes only the relevant code on request, so there is no 24/7 running server.

The full 'server' is a collection of Azure Function Apps and API Management. Each Function App resource represents a collection of individual functions, i.e., Rest endpoint, and functions, grouped by the logical separation of various services the FactServer provides

(Entity Resolution group, Feedback, etc.). To provide unified Rest API endpoint collection, subscription service, proper authentication and authorization, and much more, the API Management service is used.

API Management

API Management is a resource that serves as a gate to the FactServer. It was chosen for multiple reasons, where the most important is the 'gate' for the FactServer. To explain, API Management is a one-of-a-kind service from Azure, which serves, at minimum, as a gateway between the publicly exposed set of predefined API endpoints and the internal individual trigger endpoints from Function App. The reason to use it is that the service looks to the outside world as professional and unified, meaning that there is one base URL of the server and set of endpoints, were on the inside, since multiple Function App services are used, each has to have unique base URL. Another benefit and reason for use are defining authentication and authorization. Furthermore, it provides an out-of-the-box solution for subscription services, which is a part of FactServer's requirements.

The API Management service also fulfills the scalability and availability requirements. There is a possibility to deploy the API Management instance across multiple regions around the globe[2] and also scale by adding multiple units in one region[3]. The resource's cost varies depending on the selected tier, regions deployed, and traffic. As the service is needed and other properties satisfy all the requirements, the service is used despite the costs.

To put it together, since API Management provides the services to unify all our services and expose them to the outside world, with proper authorization and subscription services, it is one of the main and crucial resources for the whole project.

Function Apps

As already mentioned, the server functionality operates within the Azure Function App service. The server options on Azure are: (i) the IaaS solution as a virtual machine, (ii) the PaaS App Services offering, and (iii) the PaaS Function App offering. In our case, the IaaS virtual machine solution is neither feasible nor ideal solution, as it provides a significant overhead and does not have out-of-the-box security and scalability. On the other hand, App Services is a valid option, providing a runtime environment where a monolith server or microservice can be deployed, and scaled. Furthermore, Azure Functions is a serverless solution based on triggers. It also provides excellent scalability and distribution by having small individual functions, which can run independently and on-demand. The last feasible option can be a containerized application. Although it is also a feasible solution, it represents a similar overhead regarding software updates and other maintenance tasks. Therefore, at the current stage and requirements, the PaaS solution of Function App provides more significant benefits than containers.

[2] https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-deploy-multi-region

[3] https://docs.microsoft.com/en-us/azure/api-management/upgrade-and-scale

As the architecture design aims to be cloud-native and leverage all possible benefits, a Function App service was chosen. There are multiple benefits for a new application aimed at Azure to build the server logic on Function App than on the App Service. Microsoft also recommends how to choose the correct service, where if we follow the decision tree[4] for a new application with short-lived event-driven processes, the use of the Function App is recommended. Apart from recommendation, the Function App provides the benefits of easy development for the platform, support for multiple programming languages, out-of-the-box auto scalability. The number of servers increases or decreases automatically based on incoming requests and the average response time of requests.

To sum it up, the Azure Function App provides the most benefits and suits the application needs fully. It is designed by using multiple Function App services (as depicted in Figure 4.2), where the set of functions (meaning the individual endpoints) are grouped logically into groups. This approach provides easier code development, as the whole server is split into multiple smaller groups to provide easier orientation, readability, and maintainability. For system administrators, it provides easier access control (RBAC) for the developers, where access can be given to specific resources – function groups. This approach is also supported by using API Management as our gateway. Otherwise, it would bring a bigger overhead to have multiple outside-facing base URLs and more authentication and authorization setup effort. Another motivating factor, instead of the monolith server deployed to App Service, is that the Function App is more flexible and simplifies development and maintenance due to relatively short code pieces made by the developer.

4.1.4. Data Store Resource Group

The Data Source Resource Group collects multiple resources responsible for the data storage solution designed and recommended for the FactCheck++. The collection consists of two main parts, clean storage and raw storage, where the housekeeping resource interconnects both main parts.

The clean data storage contains valid new pre-processed data delivered in structure for direct consumption by services (IdaFix, Dashboard App, etc.). Simple or no additional views, functions, or stored procedures are used to obtain the requested dataset. On the other hand, raw data is "everything" else that needs to be stored. More precisely, it consists of new incoming data (from the crawler, expert users, feedback, and other inputs), archived data from the clean storage, and any additional needed data (e.g., pictures or static content).

Housekeeping resource is the connection point between raw and clean storage. The connection point is the Azure Data Factory, a data pipeline service responsible for transferring and transforming data from raw storage to clean and storing historical data from clean to raw.

The FactCheck++ is designed to process a large quantity of data for its function. This

[4] https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/compute-decision-tree

large quantity of data follows a life cycle. Clean data is present for consumption by our connected services. When expired, clean data is archived for a certain time and removed when no longer needed. Raw data is processed into clean data. Raw data can be archived if needed for data analysis purposes.

Due to data size, performance, and storage concerns, using a single database or a single resource is not feasible. Therefore, the options of splitting the data storage were examined. There are multiple ways of splitting the system into smaller parts – individual database resources. The examination results show that the most logical and cost-effective is to separate data into clean and raw groups. This option enables different resources from Azure for clean and raw data to further optimize the system, reduce storage capacity limits and make the solution more cost-effective. The split creates an "ecology chain in the cloud storage" [43], creating consumer and producer spaces interconnected only with the housekeeping resource, keeping the rest of their lifecycle separated [43]. Similar data cycles and data ingestion pipelines are widely in use by production systems (an example from the Azure Architecture Center[5]), where most of the uses are for the IoT and sensor data input, data streaming, on-the-fly processing [14], or, as in our case, the regular data loads or data processing. The data cycle is designed to keep the load balance in the cloud system and keep the data volumes and complexity at a reasonable level to avoid crossing any threshold or performance issues [43].

Raw Storage

The raw storage, as stated above, is a mixed data store for all data that is not directly consumed by other services. Based on the Ecology Chain [43], it is a storage producer system. Since not directly accessed by the consumers, the idea of this storage is to be very cheap for a large quantity of mainly unstructured data, where the performance is a secondary concern, as the data access is not frequent. This description perfectly matches Azure Blob Storage, which is included in Storage Account service [39]. Exactly, Storage Account with the Azure Data Lake Gen2 (ADSL) option, since it provides better connectivity for data analytics services and provides folder structure.

The storage account choice is obvious for a simple reason – storing large amounts of data at a low cost and archiving the historical data. Another option that can be used is a file server. However, it does not provide the connectivity out-of-the-box as the storage account does to other services in the data pipeline and poses an overhead in terms of setup and maintenance. Furthermore, the option of using the CosmosDB for this task is also not feasible due to cost and data limits in individual spaces[6], and lastly, the concept of CosmosDB is not a data archive.

The proposed architecture is to have one ADSL Storage Account with multiple containers. The container will serve as a 'database' by dividing the data by logical spaces, like Archive container, Crawler data container, incoming Feedback data, etc. Each of these containers can be organized in a logical folder structure. As the storage account

[5] https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/medical-data-storage
[6] https://docs.microsoft.com/en-us/azure/cosmos-db/concepts-limits

provides different storage tiers (Hot, Cool, Archive), these are utilized to influence the cost structure of the resource.

The scalability and replication of the data are also not an issue. Each storage account can be set up to different availability modes, meaning that, if needed, contents are replicated automatically. There are options to do this within one or across multiple data centers[7]. However, since utilized as data backup, the performance is not a primary concern as the redundant system is in the background.

Clean Storage

The clean storage represents, as mentioned, the actual data directly consumed by other parts of the system (middleware, front end) and in a format optimized for these systems. Pre-processing the data to the clean storage means the system should be performant and cost-optimized. By having pre-processed data in shape optimized for other systems, the need for query joins, stored procedures, or complex data transformation by either IdaFix or FactServer is minimized. Furthermore, such optimization means that the system can process requests faster, therefore cheaper, by minimizing the runtime, which is especially relevant for CosmosDB, as it is charged by the request units[8] (RUs), resulting in overall lower RU amount needed, thus saving the costs.

How to store the data? First, we need to know what kind of data we want to store and choose the most appropriate solution. FactCheck++ data are mostly JSON documents stored in XBase, so we are looking for a NoSQL document database. For a few reasons, the NoSQL database for clean data is better than the Storage Account or ADLS. Firstly, we need to query the data document collection, wherein the storage account we need to know the file's exact location and then request it. However, the front end and middleware services require the query functionality to obtain the correct data, usually joined from multiple documents without knowing the storage path. One may argue that in clean storage, we pre-processed the data for direct use, which is true, but it does not eliminate the querying, it is just supposed to simplify it by creating a clear structure and nesting data to eliminate unnecessary joins later on. Therefore, we need the functionalities of a document database. For this task, Microsoft recommends a CosmosDB for document databases either by following the decision tree[9] or understanding of the data models[10].

So for most of the document data, a document database is the most suited solution, but there are multiple ways to go. The best resource, according to Azure recommendation, is the CosmosDB, which is a great product, ranked third as a document store and 26$^{\text{th}}$ in overall ranking at DB-Engines[11] portal. DB-Engines is a monthly updated ranking list based on popularity, and an initiative to collect and present knowledge on database management systems [11]. Another option is creating a VM and hosting a document

[7] https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy
[8] https://docs.microsoft.com/en-us/azure/cosmos-db/request-units
[9] https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-decision-tree
[10] https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-overview
[11] https://db-engines.com/en/

database of choice in it. However, this is not the best approach, especially with the maintenance and performance, since this solution is not distributed and scaled by default. This approach (CouchDB hosted in Azure VM) is not feasible. It was determined as part of a student project within the FactCheck++ project. Therefore, due to existing knowledge and the further one gained on CosmosDB, it was chosen as the clean storage solution for any document and graph database needs. However, the cost for the resource is steep, mainly with a large amount of data and complex queries, which led the initiative to look for alternate solutions, resulting in the split of raw and clean data.

As can be seen in Figure 4.2, there are multiple resources on the clean side of the Data Store Resource Group. The XBase CosmosDB choice was already discussed. Furthermore, a second CosmosDB graph database (Gremlin API) is already proposed for UserBase, where the data is stored as a graph.

Lastly, a currently known need was a key-value store for storing user preferences. Here, key-value pairs can be easily and cheaply stored in Azure Table Storage. Table Storage is part of the storage account, providing storage for data in tabular format, easily queried or connected to other services for a low cost. Another provider of eventually same service is CosmosDB, as it also supports the table storage with the same API. If the table needs to be distributed or heavily used, the CosmosDB's higher cost will pay off. However, the Table Storage from Storage Account is more suitable at this stage.

In the future, there might be a need to expand the clean store services to different service types, like a relational database, graph, or any other store. As it can be observed, this model poses no restrictions, since Azure provides all storage solutions that could be needed, so the outlook is to follow the guide on understanding the data model and decision tree mentioned earlier, to choose the most suitable service. The integration with other services would be simple since FactServer can consume the data either by Function bindings or by calling the respective API. From the other side of the data pipeline, a connection to housekeeping (Azure Data Factory) can be initiated to almost any system as Azure provides connectors for their services and also 3rd party ones[12].

Data Management and Analytics (Housekeeping)

The Data Management and Analytics resource, referred to as "housekeeping" for simplification purposes, is the connection point between raw and clean storage space. As the intention is to extract, transform and load (ETL) data from one side to another, we are looking for an ETL tool. There are many ETL tools available on the market. The Azure offering is the Azure Data Factory (ADF), which is, as they call it, Code-Free ETL as a Service[13]. The Azure Data Factory can connect to all, in FactCheck++, used services and transform and move data. The important thing is that it is cloud-native, meaning it provides us with the needed scalability and, therefore, it can work on large data sets. For the data transformation part, ADF can do simple transformations independently. However, as foreseen in the FactCheck++, more advanced ones need to be used. For

[12]https://docs.microsoft.com/en-us/azure/data-factory/copy-activity-overview
[13]https://docs.microsoft.com/en-us/azure/data-factory/introduction

advanced needs, the ADF provides an option to connect other services like DataBricks, Azure Synapse, Hadoop Map-Reduce, and others[14]. Any of these connected services can be used. However, the most promising one is DataBricks, which is placed in the architecture diagram as a linked service for ADF. Of course, the exact use and needs might change in further development. Still, in the current state, the DataBricks notebooks seem to provide the transformation functionality needed and open another possibility on how to build dashboards and do data analytics on the data, therefore another access point to the system for data analytics. The cost of the DataBricks cluster is high, but if the cluster is not running 24/7, the cost is manageable, especially for easy data analysis.

Data Flow

As the Data Store component of FactCheck++ was described, a detailed data flow overview will be provided to understand further the concept of two storage system with Data Management and Analytics (Housekeeping) task in-between.

Data Flow from Data Producer to Data Consumer For better visualization, the data flow is represented in Figure 4.3. This flow represents the proposition of standard data flow in the system. The flow follows a clean path and one-way direction. This design ensures that data is not mixed, pipeline execution can be optimized, and any data analysis will always run on the whole data set per execution period. The numbers represent in the flow below corresponds to the numbers in Figure 4.3:

1. Data is written to respective container and folder in Raw Storage by data producer (data from Crawler or FactServer, e.g., feedback data).

2. When triggered, Azure Data Factory (Housekeeping resource) will connect to the Raw Storage, access the respective container and folder, and ingest the data from Raw Storage for further processing.

3. Data Transformation action of the pipeline. *Optional:* If needed, compute type linked service might be used to run advanced data transformation or data analysis on the data.

4. Transformed data is loaded to the corresponding Clean Storage database. ADF will open the connection and store the data to the corresponding location.

5. Data can be consumed from Clean Storage by data consumer (FactServer read functions).

[14]https://docs.microsoft.com/en-us/azure/data-factory/transform-dataexternal-transformations

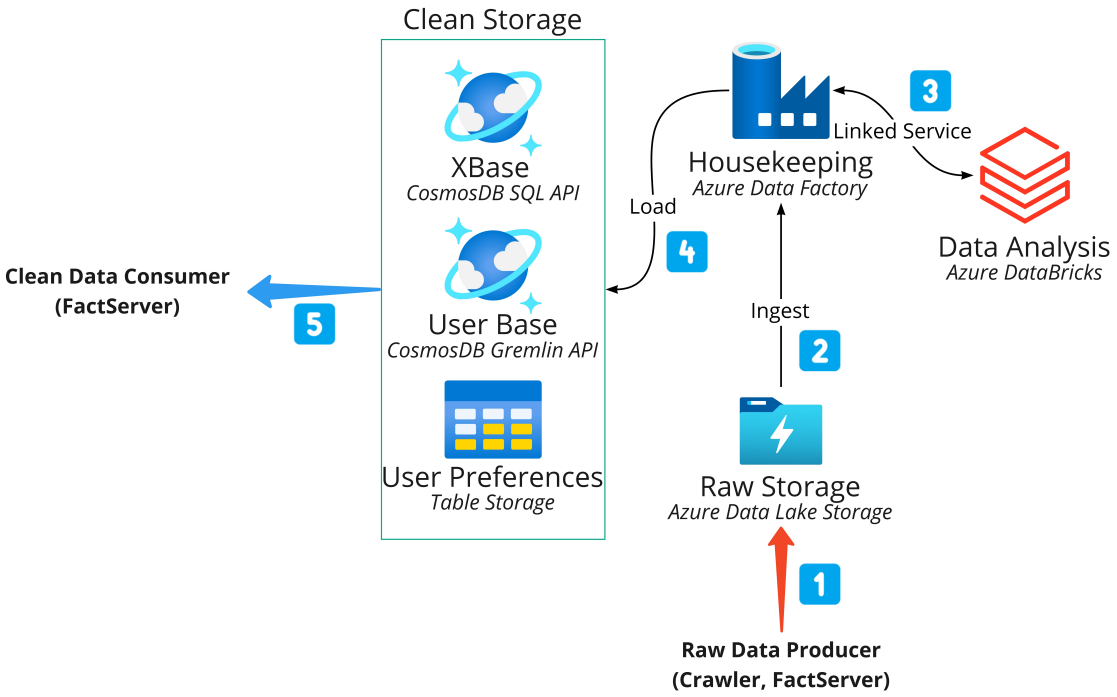


Figure 4.3.: Data Flow from Consumer to Producer

Data Flow of Archiving For better visualization, the data flow is represented in Figure 4.4. The Archive flow is used to extract old or not needed data from Clean Storage, to optimize costs and performance, and save the data for a certain time in an Archive container. The numbers represent in the flow below corresponds to the numbers in Figure 4.4:

1. When archiving pipeline is triggered, data is ingested from the Clean Storage based on set criteria (e.g., creation date)

2. Data is transformed if needed. *Optional:* If needed, compute type linked service can be used for advanced data transformation or data analysis tasks on the data.

3. Transformed data is loaded to the Raw Storage – Archive container to the defined folder. The storage file access tier will be set, based on how often will be the archive files accessed, the default is "Cool Tier".

4.1.5. Crawler Resource Group

The crawler resource group is a space for web crawler resources. Since there is no concrete application specification (like language, runtime environment, requirements), the options for the concrete resource are numerous, from Function App, through Containers to VMs.

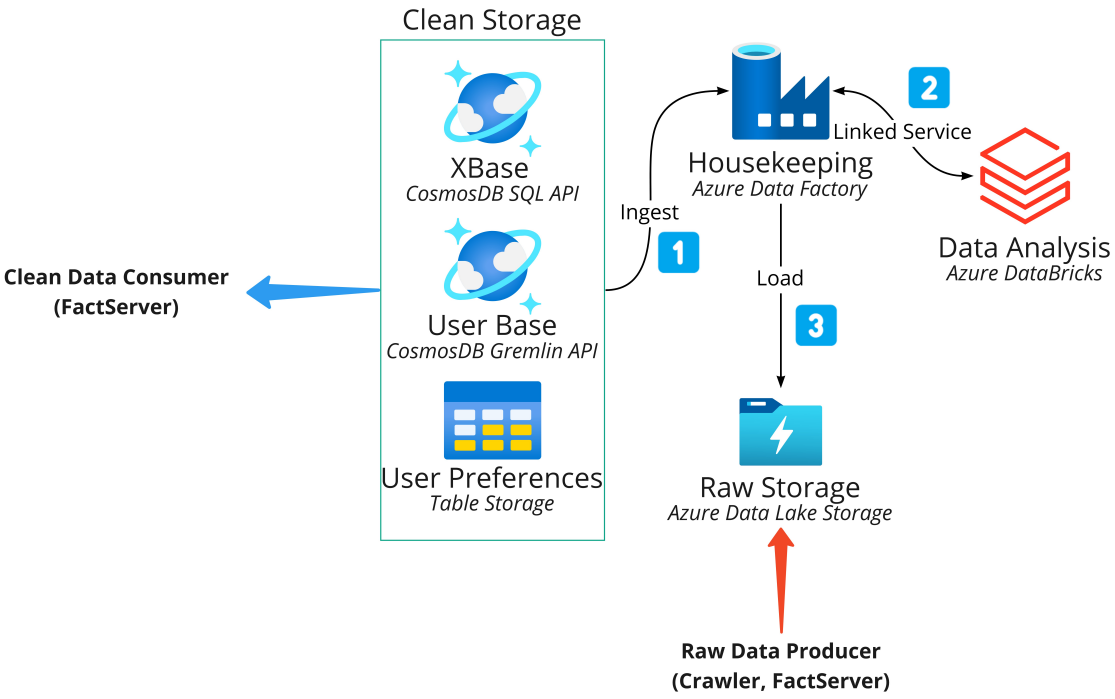


Figure 4.4.: Data Flow of Data Archiving

As there is no concrete decision on crawler resources, an interesting blog[15] on building crawler on Azure PaaS solutions was found. The reasoning was that the author wanted a scalable solution and pay-per-use. Therefore, the motivation was to use the Azure services and not VMs [26]. Moreover, the author wanted to use the Azure Function App for running puppeteer[16], which at the time of writing this blog was not possible [26]. However, it should not be a problem nowadays, since there are possibilities to extend the base Linux containers for Function App when using Function Core Tools[17]. To sum it up, either by use of function or containers, it is interesting to see a running crawler on a scalable, pay as you go PaaS solution, also benefiting from the connectivity to other used services. Therefore, the recommendation from this thesis is to follow a similar route since the benefits of scalability, integration, and cost are present.

4.1.6. Infrastructure

Infrastructure is more collection than resource group on its own. However, as also present in many Azure architecture diagrams, the infrastructure services needed are present in one

[15] https://dibranmulder.github.io/2019/02/15/Building-a-Web-Scraper-in-Azure/
[16] a NodeJS library to Chromium API, which can be used for crawling, https://github.com/puppeteer/puppeteer
[17] https://docs.microsoft.com/en-us/azure/azure-functions/functions-core-tools-reference?tabs=v2#func-azure-functionapp-publish

group, and therefore the same logic is followed here. The infrastructure section contains all identified services used in addition. Azure Active Directory, Azure Active Directory B2C, Application Insights, Azure Monitor, Key Vault, Security Center, Alerts from left to right.

The Azure Active Directory (AAD) is a necessary service, where it runs under tenant and is used for user management. The idea is to run the whole FactCheck++ project under the tenant and then leverage the Active Directory for RBAC and user control for project contributors. The Azure Active Directory B2C or business to clients is used for external user management. External users are the application's end-users (Standard or Dashboard users). The AAD B2C provides the entire user management (register, login, roles, etc.) where Azure applications can be directly connected to it, ensuring user authentication with minimal setup effort. There is a parallel running project focusing on AAD and AAD B2C.

The Application Insights is a resource dedicated to collecting telemetry of applications. It can be integrated with our solutions (API Management, App Services, Function Apps) to provide metrics about the systems containing the availability, exceptions, performance, logs, and much more. The service is for free. Only the storage of logs and use of log analytics is paid for a small fee, which is a plus. The service is needed in any larger system, since it collects all information in one place, offering a simple Logstash functionality and the possibility to drill down through individual requests across all used services.

The Alerts are used with the Application Insights. The recommendation is to set rules, e.g., five failed requests in 20 minutes. When the rule is triggered, the Alert notifies the responsible person via Email or Phone or sends the message to some HTTP endpoint. The alerts are charged by how many times they are triggered and sent notifications. However, the benefit is that responsible persons are notified in real-time that the service is having problems.

The Azure Monitor is the default service available directly in Azure Portal. The Azure Monitor collects telemetry for each service like availability, memory use, CPU load, and runtime environment-based telemetry. This data, available by default, can be used to analyze and evaluate the sizes and settings of the individual services.

The Security Center is a default service providing users security recommendations and compliance with security policies. The free tier is available by default on all subscriptions, providing continuous assessments on policies for deployed resources. For more advanced scans and management, the Azure Defender needs to be turned on, which is a paid service per server or resource [22].

The Key Vault service is planned here in case there is a need to store passwords or access keys or certificates securely within the project. Azure mainly uses managed identity

principle to eliminate the use of passwords and secrets, but not all resources support it yet. Furthermore, if there is a need to connect to any third-party services, this is the service to store the keys and securely provide them in the application.

4.2. Infrastructure – ARM Templates

For architecture implementation, ARM templates are created to implement the current described architecture, apart from the architectural diagram and other design decisions described above. The idea is that these templates will serve as the first version and be extended if any new resource will be added to the FactCheck++ system or by extending the existing ones if any additional settings or changes need to be adjusted. This template base will not only serve as a template for deploying FactCheck++ environment to future productive or test environments and as part of disaster recovery but also for other project developers to deploy either full environment or part of it in their resource groups as a sandbox[18] environment. The benefit and main argumentation point of using ARM templates is the guarantee of consistent results, so each resource or environment created by template will be always same, which ensures that developed code will work on test and production environment, since the infrastructure itself and its settings are same [30].

As ARM templates can be treated as a code, any code can be implemented in multiple ways. Whereas the format and individual resource declaration properties are prescribed from Microsoft Azure, the structure of the template or linked templates is up to the developer. Moreover, the variants of deployment options or parameter declaration can steer the design of templates. As the design of the templates can be adjusted, the following section will describe the design decisions taken to structure templates and parameters.

ARM Templates Structure

Templates can be structured by placing all resources in one file or for easier readability and maintainability in multiple files. To still have the ability of "one-click" deployment, these template files can be linked in one main template. [30]

Since the goal of FactCheck++ templates was to deploy the resources in either modular way – by individual instances or sections – or the system as a whole, they have a modular structure. This modular approach provides simplified templates since they are less complex and can deploy each resource individually or whole architecture. The structure was designed alongside the general architecture diagram depicted in Figure 4.2. As system architecture is divided into three main parts (back end, middleware, back end), ARM templates are structured the same way. ARM templates are split into three main sections with the same names. Each section contains a Main template and individual templates for individual resources. This setup can be observed in Figure 4.5. The top-level element is Main Template. The Main Template links the other three parts mentioned above. In

[18] A test environment not connected to any production or test environments of project, used solely by a developer for development purposes

other words, there is a tree structure, where Main Template is the root connecting three sections main templates as nodes and resource templates are individual leaves.

In the same figure can be observed that to the above rule, there is an exception for front end, where is no Main template. The reason is simple: there is only one resource in the current version, and therefore, an additional template is unnecessary. However, the Main template might be added in future versions if more resources are present in this section.

Other sections cohere to the structure mentioned above. Apart from the structure itself, the Figure 4.5 depicts links between the individual templates. These are depicted as an arrow with the text "linked". The latter with an arrow labeled "Depends On". These dependencies are introduced due to design requirements of Azure resources, e.g., a Function App requires a Storage Account which needs to be created before Function App deployment. Such restrictions need to be considered by designing the templates, linking, and dependencies. Other options of ARM templates can be used as well. In FactCheck++ structure, also template output is used. Output can provide some result values to deployment outputs seen, e.g., in Azure Portal, but also as input in another, dependent resource. This link is depicted with the arrow "Output".

Output Template output is designed once in the middleware section. The output goes from the Application Insights template and is consumed by the API Management template. The reason for usage is simplifying post-deployment setup since API Management will be deployed with connection to Application Insights.

Depends On In the FactCheck++ templates are multiple dependencies between individual files or sections (section Main template). These dependencies, when enforced, ensure successful deployment if one of the Main templates is chosen. In addition, these dependencies are designed to ensure the correct order of deployment is used. In the templates structure diagram, these dependencies set the deployment order, where firstly the back end section is deployment, then middleware, and lastly, front end.

The reasoning is that middleware is dependent on the back end due to the need for a storage account for FactServer (Azure Function App), and Raw Storage used for FactServer is deployed with back end. On the other hand, the front end is dependent on middleware due to App Service Plan, as FactServer and Dashboarding App have their own ASPs, the order of deployment needs to be followed if deployed in one Resource Group, as in one Resource Group, ASP for Function App needs to be deployed first due to its settings.

Between resources in one section are generally no dependencies in deployment order. Only one is middleware, where API Management is dependent on Application Insights output.

Links Links represent which templates are linked. Section main templates are linked with the Main template, and resource templates are linked to the section main template.

These links serve in connecting resource templates in a single point for bulk or system-wide deployment and as a link alongside which parameter values can be passed.

To sum the structure up, it is designed in a modular way. This modular approach provides benefits of breaking the templates into smaller, easier maintainable code pieces and allows for deployment of single infrastructure components. The modular approach is supported and recommended by Microsoft Azure – break templates into smaller, reusable components and link them together at deployment time[30], which is recommended for advanced scenarios. Linked templates enable us to break down the solution into targeted components[29]. The granularity used is one template per resource, allowing for very targeted deployment of only needed resources, which still adhere to the setup. Template links then connect these on two levels. The two-level setup is introduced for greater flexibility for the developer. The working assumption is that in the project's current state, the templates will mainly serve for student-developers to create their sandbox environment inside their resource group (i.e., student space). However, since the student projects are usually focused on a single system section, they do not need to create a full sandbox environment, instead only the section required for their topic, thus saving cost by not deploying unnecessary components. The other supporting argument for the template setup is a plan to host a stable version of common resources (e.g., FactServer, Housekeeping ADF, Raw and Clean Storage, etc.) in one environment (e.g., test environment). These resources can be utilized by a project that needs the data or connection to those services but does not work on them. For example, a Dashboarding App project needs the data and API endpoints of FactServer but is not changing anything in FactServer. In this scenario, the student utilizes a test system, FactServer, thus deploying only front end resources, which is easily possible by only taking either section main template or single resource template in the current design. However, at the same time, full test environment can be deployed from the Main template.

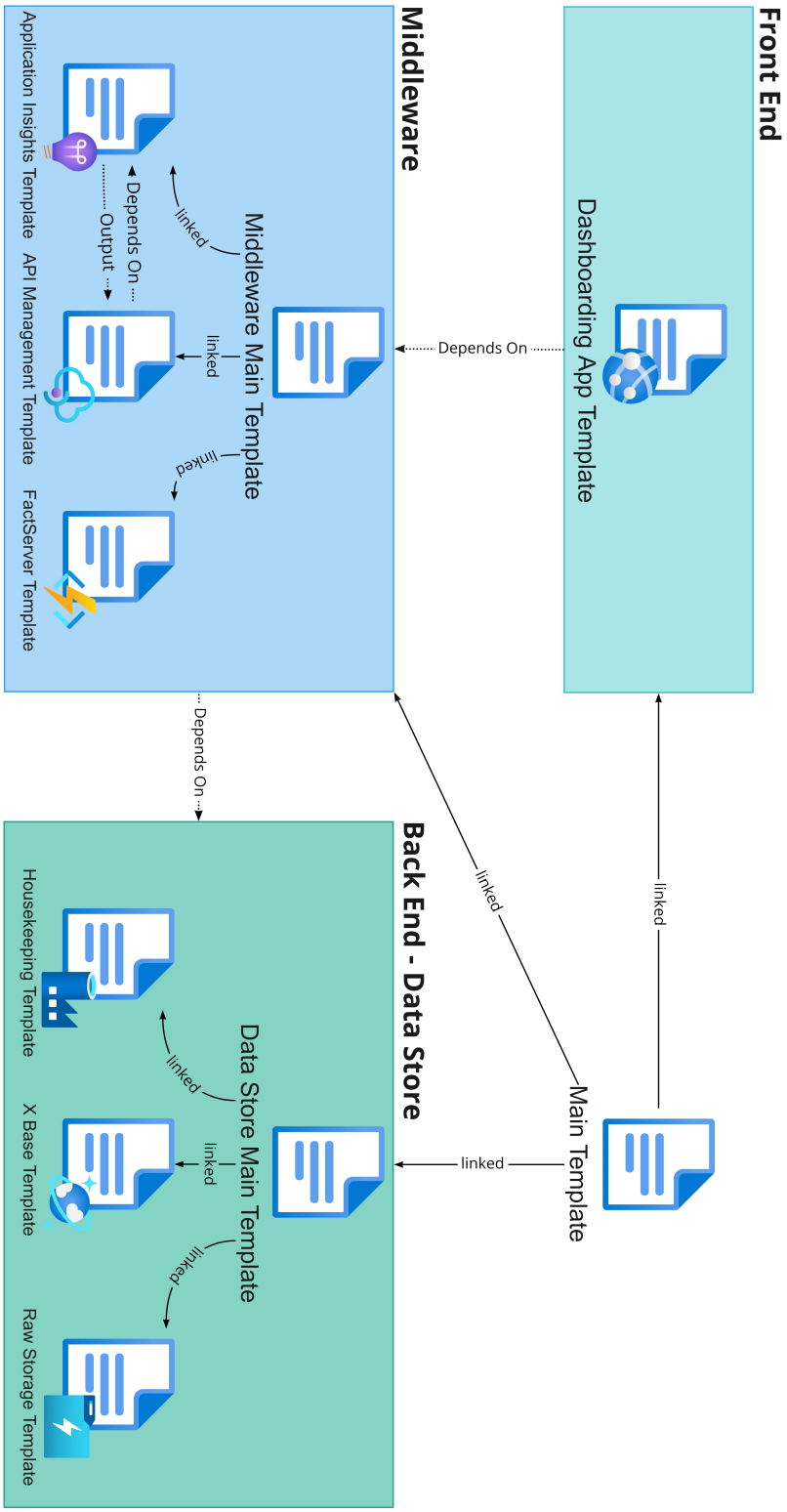


Figure 4.5.: ARM Template Structure Diagram

5. Implementation

As the architecture design is known now, let us discuss how to get the working Azure instances in a few simple steps. The implementation of the project architecture consists of two parts: the concrete selection and setup of the services (choosing an operating system, size, CosmosDB API, etc.), and second is the actual deployment of the chosen services and any manual setup needed. Accordingly, this chapter will be structured. Firstly, the selection of the systems will be presented, and after the Azure Resource Manager (ARM) templates responsible for deploying the environment will be presented.

5.1. Azure Setup

5.1.1. Front End Resource Group

The actual setup of the front end App Service resource is quite open since the requirements are not defined yet. As it will be a web application, the template is set up to allow either NodeJS or Java App Service deployment. This choice complies with the used technologies in the project. If only static content is needed (e.g. Angular Web Application), the App Service will be exchanged for Static Web App offering from Azure to serve static content.

5.1.2. FactServer Resource Group

API Management

API Management service setup, although the complexity, is straightforward. The important is a correct selection of pricing tier. A development tier is selected for development environments (i.e., sandbox, testing). For the production system, the higher tiers are intended. The actual tier depends on usage, and the step-up should be done based on monitoring data.

Manual after deployment setup is unnecessary, as the Application Insights monitoring is linked with the template during deployment. Therefore, after deployment, resource development can be started immediately.

Function Apps

The FactServer is currently designed as six individual Function Apps. The deployment template will create six same resources. The resources are numbered from zero to five. Each resource has the same configuration, a Python 3.9 serverless Function App. The language choice comes from the requirement, and existing implementation is done in an older version of Python. Due to ending support for the older Python versions, the

currently available 3.9 is used. The operating system is Linux as the only option, as Python is currently supported only on Linux. A serverless option is an obvious choice for the current state of development and beginning of production use. Suppose monitoring of the costs and usage proves the serverless option is no longer rentable. In that case, one of the production ASPs should be used (the premium selection) to have reserved compute for the system. The step direct to the premium ASP model is that the consumption-based serverless model covers basic and mild production use. If it would not be enough, a dedicated ASP needs to be used or potentially consider Durable Functions.

The manual setup after deployment is needed. The Application Insights has to be connected manually, as for operating system and language selection, this cannot be done via template deployment. This can be done by simply choosing the Application Insights resource in the Application Insights tab in the Function resource. It needs to be done for each Function App resource. A guide on how to link the Application Insights can be found in the Appendix A.2. Other manual setup might include connection to Active Directory. However, it depends on the outcomes of the ongoing project defining the authentication concept.

5.1.3. Data Store Resource Group

Raw Storage

Raw Storage consists of Azure Data Lake Storage. The "Enable hierarchical namespace" flag needs to be set to true to have the Data Lake version of Azure Storage. In ARM templates, it is `isHnsEnabled`. The rest of the template is a standard Storage Account template. Thus by changing the value of this parameter to false, a standard Storage Account will be deployed.

The selected account type for development environments is Standard LRS (locally redundant). It is the cheapest one and sufficient. There is no need for advanced replication and backup for the development environment. The recommendation is to use Standard GZRS (globally zone redundant) for the production environment. This account type means that data will be replicated globally and within availability zones. If the monitoring analysis shows speed problems, the Standard GZRS can be changed to Premium GZRS. The template sets the access tier to the default "Hot" tier. The soft delete option for files and containers is turned on by template. The default is seven days retention period, during which the accidentally deleted files or containers could be restored. Security setup is left to the default by Azure - HTTPS as default, no public access, and minimum TLS version of 1.2.

Manual setup after deployment contains the creation of containers needed with corresponding availability tiers. Here for archive containers, a "Cool" tier is recommended as these data will not be accessed often. Other tiers are "Hot" as the access will be frequent.

Clean Storage

Clean storage consists of multiple, not yet fully specified resources. The main structural point defined is XBase, a CosmosDB database. It is still in development if any other storage type will be needed apart from the document database. Thus, the implementation currently contains a single CosmosDB document database with SQL API. The serverless consumption model is selected for development environments as the pay-as-you-go model is more suited for this kind of usage. A reserved DTU (Data Transfer Unit) model is designed to be used for a production system. The amount of reserved DTUs can be changed. Therefore, it can be specified and adjusted based on monitoring data collected with production use. Consistency of database is not a big concern in FactCheck++ design, as there is defined data flow, where FactServer only reads, and the Housekeeping jobs write. As the writes are done in jobs, even with eventual consistency, the data will be available at some time (according to Azure[1]). Thus, it becomes stable after this time, as the next write will be at the next job run (e.g., next day). Due to this, a full read speed can be potentially utilized from CosmosDB. Currently, the implementation uses Session consistency as the default one. This level makes sure that within-session, write order is kept with direct visibility.

Manual set-up, apart from creating containers in the database, is not needed.

Data Management and Analytics (Housekeeping)

Housekeeping resource consist of ADF (Azure Data Factory) resource. The ADF resource is configured to the standards by default values for parameters. The setup is configured to use for encryption Microsoft managed keys, and the developer portal is accessible by a public endpoint. The use of Microsoft managed encryption keys is consistent with the approach of using PaaS solutions, meaning that management is left to Azure, the FactCheck++ team provides the code. Suppose there would be any security requirement to use any other key not managed by Microsoft. In that case, it can be easily changed in the parameters by switching `enableCMK` to true and providing key name from linked Key Vault to `cmkIdentity` parameter. The access to the developer portal from a public endpoint means that no VPN for access is needed. As there is no VNet or private network designed in the system, the access can be set to public. Authentication through Azure AD is enforced before accessing the portal.

No manual setup is needed. The resource is ready to be used after deployment.

5.1.4. Crawler Resource Group

As concrete crawler resource(s) are not yet specified, no concrete implementation is done. If the resource needed would be Function Apps as the similar project use as reference suggests, the template for FactServer can be used. The same applies if App Services are used. In the case of any other Azure resource, steps, as specified in the next section, can be used to export, modify, and add a new template to existing ones.

[1] https://docs.microsoft.com/sk-sk/azure/cosmos-db/consistency-levels

5.1.5. Infrastructure

Most of the resources used in the infrastructure section are part of Azure by default. Thus, no implementation is needed for the resource, possibly adjustment for some. For those, it is specified below, other resources are ready to use.

The Azure Active Directory (AAD) set-up for internal and external users is part of the ongoing project, therefore no set-up nor implementation of further access controls was done.

The Application Insights needs to be deployed and linked to the resources. The Application Insights does not include any unique setup apart from name and type. The type is Classic. The difference between Classic and Workspace-Based is that Workspace-Based uses an already existing Log Analytics resource, whereas Classic uses a separate one linked to the resource itself.

The Alerts are not as a resource itself. Therefore no implementation nor template is created. Instead, the rules must be specified and manually created in the Azure Portal.

The Security Center is automatically available for every resource in Azure. Therefore, manual intervention or setup is needed only if the default configuration does not comply with the security concept. Currently, the default configuration is used with no anticipation of using further options for both development and production environments.

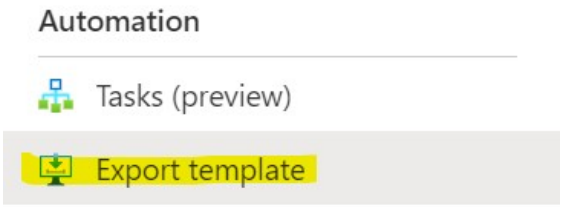
The Key Vault resource is designed as if needed, then create the resource. As currently is no need to store secrets, no resource is created and implemented. However, if there would be any need, it can be created with steps in the following section. The setup and parameters for the deployment of Key Vault are straightforward. The standard tier covers all needs in case Key Vault will be needed.

5.2. Azure Resource Manager Templates

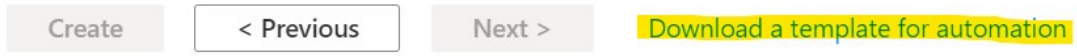
Azure Resource Manager (ARM) templates are Microsoft Azure option for Infrastructure as a Code. They allow defining custom infrastructure deployment consisting of Azure resources. The templates contain resources needed with their setup and order of deployment. Further, the following section will discuss how they are implemented, options for using and deploying the whole infrastructure, and possible further development of the templates.

5.2.1. ARM Templates Implementation

ARM Templates can be implemented in multiple ways. The design approach chosen and reasoning were discussed in Chapter 4. The actual implementation is based on this design. Same as for the template design, they can be implemented in multiple ways.



(a) Export in resource overview - option is present under "Automation" section in left navigation bar.



(b) Export from Azure Marketplace - Confirmation/Validation screen of Azure Marketplace on the bottom next to the buttons

Figure 5.1.: ARM Template Export Options in the Azure Portal

The templates can be written from scratch directly in JSON format or Bicep[2]. The JSON format is introduced in background information Chapter 2. The Bicep is a second offering from Azure. The Bicep is a domain-specific language (DSL) that uses a declarative syntax to deploy Azure resources [27]. Bicep provides a code-like structure that is more developer-friendly, essentially with a similar look and feel as Terraform[3] which AWS Cloud users widely use. Azure supports both template formats since JSON templates can be decompiled to Bicep if needed, and internally, Bicep templates are compiled to JSON format and deployed.

Another option for template creation is export from Azure. Azure Portal offers an option to export an ARM template in JSON format. There is an option to either export a template from the confirmation screen before deployment in Azure Marketplace (Figure 5.1, bottom image). The second option is to export a template from an existing resource. This is possible from the Azure resource overview, wherein the left navigation bar, there is an option "Export Template" under "Automation" section Figure 5.1, top image). Both ways, Azure export tool exports only supported settings and parameters. Moreover, the export functionality is only available for single resources. The whole system cannot be exported at the moment.

These two approaches are combined to implement the templates by exporting configured valid templates from Azure Marketplace, followed by customization and linking. The templates are exported and further developed in the original JSON format. The JSON format was chosen due to direct support by the ARM Manager without any need of (de)compilation and greater support and documentation from Microsoft. It originates from the time of the development, meaning that nowadays, Bicep can be seen as superior

[2]https://docs.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview
[3]https://www.terraform.io/

concerning development and template writing. However, it is a relatively young product from Azure, and thus it was still in Preview and has fewer functionalities during the development. Despite the recent Bicep development, the recommendation is to keep JSON templates. The reasoning is that the primary purpose of Bicep is to provide a more developer-friendly experience, not a tool with more functionality. For now, the JSON ARM template is the main way to specify resources. Therefore, it has all the released functionality.

In the FactCheck++ project, templates, as designed, are split across multiple files in tree-like logic to compensate for the readability loss by using JSON format. As depicted in Template Structure Diagram (Figure 4.5), JSON files are structured accordingly. There is one JSON file for each resource, the main JSON file, which is a linked template of resources for each section, and finally, one main JSON linked template.

Resource Implementation

All templates for the individual resources were made the same way - by exporting them from the Azure Marketplace, followed by needed modifications. The implementation was done in the following steps:

1. Finding the resource in Azure Marketplace, filling determined the options in the forms, if validation passes the template was exported.
2. The template was edited
   - For each parameter, a default value was specified (except Email and other not suitable parameters) if needed, a restriction on input was specified, and description
   - A tags array parameter was added or updated to the standard one used in the project
   - Resource name is concatenated with matriculation number from tags array
   - Parameters used in resource section were corrected if needed
3. A parameter file was created

The resource parameter values used in step 1 are the same as described in the previous section. The second step consists of editing. As described to the parameter declaration are added default values where appropriate, allowed values, and descriptions. Azure tags are used within the University Subscriptions to categorize resources further. As a result, tags are added to each resource deployed. The tags contain student name, project name, matriculation number, and supervisor name. As they are used in each template, a project-wide tags array parameter object is defined as shown in Listing 5.2. The standardization makes it easy to pass value between linked templates, and the usage is the same for easier maintainability.

The standardized parameter declaration and resource declaration templates are used for all resources. Despite standardization, a slight variation exists in the FactCheck++

Server template. In this case, the template should deploy six identical Azure Function resources. For this case, instead of writing six individual resources, a `copy`[4] function is used. This function deploys the specified number of resources. With this function, no code duplicity is needed. The usage is depicted in Listing 5.1. It is a simple function to use, as depicted, it added to the resource declaration. A name of the copy function and number of copies is required. The inputs could be parametrized, as in this case, which means that the number of copies can be adjusted by parameter. The copy function also gives us a helpful function `copyIndex()` which returns a zero-indexed serial number of the copy, which can be freely used in the resource, e.g., as part of a name. Currently, it is used as an extension to the naming.

```
"resources": [
    {
    ...
        "name": "[concat(parameters('factServerName'), copyIndex()
    , '-', parameters('tagsArray').MATRIKELNR)]",
        "copy": {
            "name": "factserverfunctiongroupcopy",
            "count": "[parameters('numberOfFunctions')]"
        },
    ...
    }
]
```

Listing 5.1: Example Usage of the Copy Function

```
"tagsArray": {
   "type": "object",
   "defaultValue": {
      "MATRIKELNR": "01234567",
      "NAME": "Last, First",
      "PROJECT": "Thesis",
      "SUPERVISOR": "LastName",
      "FACULTY": "Informatik"
   },
   "metadata": {
      "description": "Tags"
   }
}
```

Listing 5.2: Tags Array Object Declaration

In the step 2, the resource name are defined. Due to the requirement from Azure of using globally unique names for certain resources, a following default naming structure was defined.

Resource Naming Names of resources are in respective parameter files and as default values. Naming is consisting of parts: `fc` + `resource name` + `matriculation number`.

[4] https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/copy-resources

For resources that forbid the use of '-' or special characters or big letters, these are omitted, thus using a single word. Matriculation number is used to ensure that global unique name constraint is conformed for services.

The naming is used as the default name for each resource. The matriculation number is added to the name during deployment by using `concat` function, similar as shown in Listing 5.1.

Linked Templates

Section main templates and Main template are implemented as a linked template. Link template has the same structure as a normal template, as the link is defined by using a resource of type `Microsoft.Resources/deployment`. This resource definition contains a link to the template and parameters needed for the linked template. If `uri` link is used, the template has to be on a publicly accessible web location, or relative paths could be used for underlying templates under main one [29]. The parameters can be either defined or obtained from a linked parameter file. In the case of FactCheck++, a declaration of the parameters is used in linked templates. The decision is an implication of the default deployment method for the project. An example declaration of a linked template resource is depicted in Listing 5.3. The declaration of other sections (e.g., parameters) is analog to the resource templates. Deployment order function (i.e., `dependsOn` function) can be declared to specify the deployment order further.

In the linked templates, output from the underlying resource can be used as an input of the dependent resource. First, the output needs to be declared in the resource template. This output can then be extracted and used in input parameters in another template. Noteworthy is that the template using output from another as an input must have a `dependsOn` specified for ARM Manager to deploy resources in the correct order. The output can be extracted with command as presented in Listing 5.4. `reference` function accepts a string parameter which is the same as the name of the resource representing linked template with output (i.e., name parameter in Listing 5.3). `outputs` returns a list of all outputs since it can be multiple. After `outputs` a wished output can be selected using dot access and lastly, the `value` to get the actual value.

```
1 "appInsightsObject": {
2     "value": "[reference('appInsights').outputs.appInsightsObject.
    value]"
3 }
```

Listing 5.4: Input Parameter Value Declaration by Use of an Output Value from an Another Template

5.2.2. ARM Template Storage and Organization

The templates are organized in the folder structure to organize them further. There is one folder for each section, in the folder, section main template, resource templates, and parameter files. The templates are versioned using Git version control within the

```
"resources": [
    {
    "type": "Microsoft.Resources/deployments",
    "apiVersion": "2020-10-01",
    "name": "appInsights",
    "properties": {
        "mode": "Incremental",
        "templateLink": {
            "contentVersion": "1.0.0.0",
            "uri": "https://fctemplates.blob.core.windows.net/
   templates/ARM%20templates/middleware/
   applicationInsightsTemplate.json"
            },
            "parameters": {
                "type": {
                    "value": "[parameters('type')]"
                },
                "appInsightsName": {
                    "value": "[parameters('appInsightsName')]"
                },
                "regionId": {
                    "value": "[parameters('regionId')]"
                },
                "tagsArray": {
                    "value": "[parameters('tagsArray')]"
                },
                "requestSource": {
                    "value": "[parameters('requestSource')]"
                }
            }
        }
    }
]
```

Listing 5.3: Linked Template Resource Declaration

joint project repository. This project repository is private, which collides with the ARM Manager requirements of public template locations to deploy linked templates. Therefore, the actual templates used are currently stored in separate Azure Blob Storage used for this purpose. In the storage container, the templates are stored in the same structure as in Git. The advantage of a public Git repository (e.g., GitHub) is that the templates are not easily searchable. However, the disadvantage is that they are not protected, and with every change, they need to be re-uploaded to the storage. This approach was used to simplify access to the templates by other team members, as there is no need for having a local copy of the templates. Furthermore, everyone is always building from the same master templates.

5.2.3. ARM Template Deployment

Template deployment is realized via the "Deploy to Azure" button deployment method. This method is chosen among others due to its simplicity for users, as it does not require any prior knowledge of ARM templates and how to use them. This option also allows modularization, where for each resource template, section main templates, and Main template, an own button is created. Thus, the deployment of environment can be targeted at wanted resources only.

If the "Deploy to Azure" button deployment method is not available, or it cannot be used, the templates can be deployed by other template deployment methods as described in Chapter 2, Section 2.6.5.

6. Experimental Illustration of the FactCheck++ Proposed Architecture

This chapter will discuss experimental illustration of the proposed architecture and ARM template implementation. As the FactCheck++ project is not yet fully available on the Azure platform, the proposed architecture will be compared against the Microsoft Architecture Center's relevant architectures. Finally, other students' work using Azure and architectural concepts from the proposed architecture will be presented.

The ARM templates are evaluated by deploying them and checking the results. In addition, feedback from other students was collected and will be presented in this chapter.

6.1. Experimental Illustration

The proposed architecture is a cloud-native system architecture complying with all requirements of the FactCheck++ project. The truthfulness of the sentence above is challenged in the following sections.

6.1.1. Cloud-native Architecture

The proposed architecture, as depicted in the Figure 4.2 and described in the previous chapter, is a cloud-native architecture. Therefore, if FactCheck++ follows the architecture, it can be considered a cloud-native application. According to [34], an application is considered cloud-native if it is architected for the cloud (i.e., leverages from microservice architecture and Serverless). The proposed architecture is architected for the cloud using serverless architecture for the FactServer and cloud storage solutions. Apart from the simple service, according to the [35], there is a Twelve-Factor App[1] methodology. Developers should follow this methodology to achieve cloud optimized applications. This original list from 2011 was updated with additional factors, often referred to as 12+ [35]. As the factors are on the application level, not all of them can be considered on the architecture level. Despite that, on all fifteen factors (12+ factors) from [35] will be the proposed architecture evaluated in the following form: a factor name will be followed with short argumentation.

1. **Code Base:** A unified codebase for the whole FactCheck++ project is planned, and already a joint private GitLab project is set up and already used for ARM Templates. This GitLab project is provided under the Faculty of Computer Science, University of Vienna GitLab account.

[1] https://12factor.net/

2. **Dependencies:** This factor cannot be directly controlled by architecture, as it depends on actual project setup and implementation. However, this recommendation of having isolated dependency packages for each Function App set can be considered as a recommendation from this Thesis.

3. **Configurations:** Configurations options of the selected services are limited by design - PaaS. Therefore, the application-specific configurations should be done through the configuration section in the Azure portal or be configured during deployments by CI/CD pipeline. Thus, if the implementation follows this recommendation of using provided tools, the application will also comply with this factor.

4. **Backing Services:** The decoupling of the services is by design, as the used data store services can be either accessed by URL endpoint or by SDK where available.

5. **Build, Release, Run:** This factor concerns the best practices in CI/CD pipelines, and thus this factor is not directly controlled by the proposed architecture.

6. **Processes:** By using Function Apps, each function is an individual event execution as an individual stateless process. Thus, the architecture complies with this factor.

7. **Port Binding:** With this factor is the proposed architecture conforming by design, as each Function App instance (function bundle), which can be logically viewed as a Microservice, has its URL path. Therefore, the services are isolated.

8. **Concurrency:** The scaling of Function instances is out-of-the box, provided by the Azure.

9. **Disposability:** This factor is currently not the concern of the FactCheck++ directly, as it is provided by the Azure.

10. **Dev/Prod Parity:** The proposed architecture anticipates having multiple environments, from which at least one (e.g., test environment) has the same set-up as the production environment.

11. **Logging:** The log collection, aggregation, and further analysis are architected to be performed by the Application Insights instance.

12. **Admin Processes:** The batch processes such as data cleanup, analytics, archiving are triggered and managed within Azure Data Factory. As in the factor description, ADF can be considered the independent resource from application logic.

13. **API First:** The whole project design is based on providing every capability through a robust API.

14. **Telemetry:** This factor extends the number 11. Logging and collecting system telemetry. All this data collection and analytics is provided within Application Insights.

15. **Authentication/Authorization:** Azure Active Directory will provide the identity and access management for internal access to the resources itself and by Azure Active Directory B2C for customers. The exact implementation and details are the subject of ongoing research.

In conclusion, we may say that the proposed architecture is cloud-native system architecture. Going further, if the development relevant factors will be implemented as recommended, the resulting application will also be cloud-native. Being cloud-native architecture, it satisfies most of the FactCheck++ regarding cloud architecture, performance, and scalability.

6.1.2. Design Concepts

In the following, a design concepts used will be compared to existing Azure architectures from the Microsoft Azure Architecture Center. This is done to illustrate, that the architecture is sound and well aligned with the industry standard of the time.

Overall The overall design follows the layered design pattern[2]. The layers are as follows: Presentation layer is the IdaFix extension and the Dashboard, Application/Business layer is the Middleware (FactServer), and Repository layer is the Data Store (XBase, Raw Storage). By using the layered pattern, the architecture follows sound industry-wide accepted principles of software design, in this instance, the design of web applications and API servers. The actual use of the cloud does not negate the design patterns, as they still apply on the theoretical level.

Middleware The Middleware section consists of the API Management and FactServer, a collection of functions. These two services joined create the well-defined collection of APIs, where API Management centrally exposes the collections of APIs endpoints and handles authentication. The Function App collections consist of the business logic of the FactCheck++. The business logic (individual functions) are designed as short-lived stateless processes triggered by HTTP calls (i.e., events). The selection of Function App is the best choice for such workload, based on the decision tree[3] provided by Azure. The combination of API Management as a central resource that presents the API collections in a unified manner and multiple Function Apps is standard practice. An Example of such use is presented in "Loosely coupled quantum computing"[4] architecture provided in the Architecture Center. In this example can be observed that API Management is analog to the proposed architecture, used unified front (gateway) and authentication endpoint. The correct functions are then triggered from the API Management with all required input parameters.

[2] https://www.baeldung.com/cs/layered-architecture
[3] https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/compute-decision-tree
[4] https://docs.microsoft.com/en-us/azure/architecture/example-scenario/quantum/loosely-coupled-quantum-computing-job

To conclude, the choice of Function App and API Management is backed up by the use recommendation of the services and real industry used architecture.

Data Store Data Store design consists of cloud recommended solutions. The storage consists of two storage resources, the CosmosDB for fast and easily queried access to the actual data and the larger Data Lake Storage Account for storing raw data, archived data, or any other files. The CosmosDB is a cloud base NoSQL implementation from Azure, a recommended solution for the document store [9] and selection of these storage resources for selected tasks corresponds with a decision tree presented in [8]. The pipeline systems (the Housekeeping - ADF) as ingestion pipeline or between storage section is common practice in cloud, especially in Big Data applications. Such architecture can be observed in "Medial data storage solutions"[5] Architecture from Azure Architecture Center. Analogous to the proposed architecture, the data are first ingested and stored in the Data Lake and later analyzed [10]. The difference is in the placement of ADF resource, as the requirements are not the same. Despite that, it shows that using a split system is standard practice, and selected resources correspond to the recommended use and are well suited in cloud-native apps.

6.1.3. Existing Azure Implementations

Some sections of the FactCheck++ project are already partially developed within the Azure ecosystem. These sections are FactServer selected functionality, API Management - Subscription functionality, and FeedbackBase. The FactServer uses Function Apps with API Management. From the feedback received, this combination also works in practice and was confirmed successful experimental implementation. The FeedbackBase is the first project using the proposed data store architecture. From the feedback collected, a bit tricky part is the ADF, as there are some limitations in terms of loop limitations and especially the costs. The loop limitations could be overcome by using DataBricks jobs, as discussed in the Design chapter. The higher cost was expected based on the pricing structure. The steeper price during development and testing is due to frequent use of the pipeline, where the cost will be stable by the use of scheduled runs. Another remark was to the development is the duration of redeployment, where there are about 5 minutes wait time, making the development slower.

Despite the remarks to the ADF, it is currently the best tool offered by Azure for the task. After feedback considerations, the recommendation for future work is to collect more experience and knowledge in the tool and consider using DataBricks or Synapse linked computing services to run the workload. Using, for example, DataBricks with Python provides a user experience similar to Jupyter notebooks[6], which provide faster code compilations and easier debug options for developers. With this approach, logic can be implemented in Python and then deployed to the ADF pipeline, reducing ADF pipeline runs and redeployment, thus lowering the cost and time needed.

[5] https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/medical-data-storage
[6] https://jupyter.org/

The feedback collected on the storage separation model (raw and clean storage) was very positive from the FeedbackBase project. According to the received information, it was proven in practice in an experimental implementation as a suitable model which allows storing original data for later analysis and providing fast accessible results ready to be consumed by the FactServer.

The feedback was collected throughout the thesis through various channels such as FactCheck++ project workshops, direct messages, and meetings.

6.2. ARM Template Evaluation

The ARM template evaluation was performed by testing the implemented templates. In addition, tests were performed within the FactCheck++ team. The feedback was also collected and will be discussed after evaluation.

6.2.1. Evaluation

Design The ARM templates are designed using linked templates. This is the recommended approach of design as it breaks down the templates into separate smaller templates, easier to understand templates, which can be independently maintained and deployed, thus lowering maintenance efforts [30]. Using this approach, the FactCheck++ templates benefit from the possibility of deploying specific sections or resources. The benefit is for the newcomers, as they can easily set up their sandbox environment only with needed resources and still benefit from a standardized environment.

Tests The testing was performed manually by deploying the ARM templates. The following tests were performed:

- Deployment of each template into an empty resource group
- Deployment of each template into deployed resource group (contains all resources from the Main template)
- Incremental deployment into empty resource group (incremental deployment of resources and sections)
- Deployment of resources into fully deployed resource group with different parameters (resource should be updated)

The following deployment options were tested:

- "Deploy to Azure" Button
- Azure CLI/PS deployment

Once this comprehensive testing was successfully finished, the templates were offered to the other team members to try them. Testing performed was the deployment of template or multiple templates through the "Deploy to Azure" button. The results and feedback were collected in the form of direct messages and the simple feedback form provided.

6.2.2. Evaluation of Feedback

The feedback was collected through direct messages, comments in team posts, and a feedback form. The feedback was collected mostly in an informal way with direct interaction with the responders, as most feedback was received through this channel. From that can be concluded that the responders preferred direct informal feedback. The informal feedback collection was preferred due to simplified collection, less effort from responders needed to provide feedback, and a relaxed environment with an option for asking further questions. Alongside the informal way, also a formal way was provided via the feedback form with a few simple questions listed in the Appendix A.4. Both feedback channels were analyzed together, and the results are discussed below. The feedback analysis was done by summarizing the pros and cons mentioned, where valid repeating concerns and suggestions are reflected as future work.

The overall response and feedback to the templates were very positive. The "Deploy to Azure" button's selected approach was proven to be correct for beginners in Azure or ARM templates. According to team members, the templates are easy to use, and the "Deploy to Azure" button deployment method is convenient.

The separation of the templates was taken well as well. As reported, it provides flexibility with deployments and possible redeployment, if needed.

The deployment process consists of two main steps - parameter input and deployment. To the parameter input step, mixed feedback was received, whereas, on the one hand, it is easy to use as it is user-friendly display and input method, the majority of values provided as default values, easy setup experimentation due to available parameters, and that only minimum of values needs to be provided by a user. On the other hand, the critique was expressed to an overwhelming amount of parameter input fields observed at first glance. Also, the feel of missing description was mentioned, which might raise some questions for new joiners. As an output from this mixed review can be observed that for non-specialists, it would be easier to have only necessary input parameters shown. However, advanced users appreciate the optionality of experimenting with the setup. As the ARM templates should be controlled mainly by parameters and not hard-coded values [30], the future task would be to add a description for other parameters and review guide if simplification or further information would ease the use of the currently preferred deploy option.

To sum up the feedback, it was positive, with a couple of valid improvements points reflected in the Future Work section.

7. Discussion and Conclusion

This last chapter will provide discussion and reflection of the work, outlook for the future and conclusion of the thesis. The chapter is separated into the respective sections.

7.1. Discussion

To sum the thesis up, the research behind the thesis was to propose a cloud-native architecture optimized for the Azure cloud. The main requirements for the architecture state that it has to be easily scalable, provide needed performance with the option to adjust performance models, be cost-efficient, and provide telemetry collection and analysis. The architecture should be showcased as an experimental implementation within Azure. The results indicate that the research effort was able to deliver expected results conforming with the set requirements. To summarize, the thesis provides a proposal of cloud-native architecture for the current FactCheck++ project. Additionally, Azure Resource Manager templates implementing the architecture are provided. The templates provide means to re-deploy the proposed architecture in any resource group. The templates are an additional deliverable, as it was not planned initially.

The results presented suggest that the provided architecture proposal meets all the requirements. The contradicting requirements were evaluated in context. For example, cost and performance requirements are always contradicting, especially in a public cloud setup, as cost correlates with the price, where more performant service is also more expensive. In such contradictions, the services proposed and their evaluation was done in the context of the FactCheck++ needs. Therefore, not only the cheapest option was always selected, instead the best resource for the task. An example can be selecting Azure Data Factory resource, which is more expensive than other solutions. As presented, it is the best for the task and has a good price to performance ratio.

In line with our cloud-native application approach is the overall selection of the resources, where the architecture design started from ground zero. This approach was chosen to bring the best results, as all decisions done were without concerning the existing on-premise implementation. This approach brought the architecture to conform to the set requirements without compromising it due to compatibility with the current on-premise implementation. The most unexpected result of this architecture is the final storage solution. At the beginning of the research was, only known the need for a NoSQL database, but after the initial study of the CosmosDB became obvious that CosmosDB with the planned amount of data to be stored is not feasible to store all of it in the CosmosDB due

to costs of the service and data amount limits. Based on the findings and comparison of other models, the split system was found to be the most reasonable and cost-effective.

The use of cloud technologies and providers builds on the existing evidence in the literature, mainly [2], [13]. It enforces our heading of hosting the FactCheck++ in the cloud as the most optimal solution for start-up projects. Without any initial hardware cost, the project can leverage state-of-the-art services provided at a reasonable cost.

The evaluation also shows that the final architecture complies with all Azure recommendations and proper use. This proves the usability and practicality of Azure and the feasibility of creating a complex cloud-native architecture composed of PaaS (Platform-as-a-Service) components. This result was expected from the theoretical Azure documentation, but it was surprising that it worked out in the small-scale experimental evaluation. Despite the successful experiments and previous research, some concerns were raised about the costs and usability of the Azure Data Factory. These concerns will be addressed in future work.

Due to the lack of literature on cloud architecture of fact-checking systems, and the lack of complete implementation of the services in the cloud, the evaluation of the study results is limited to the theoretical level and small-scale tests. With the ongoing implementation of the project, it would be possible to evaluate the architecture on a practical level as well. However, as this is currently out of the scope, a follow-up on this topic is recommended to set the actual production resource sizes and make needed adjustments to the reference architecture.

The missing implementation of the test system constraint this study to provide any relevant cost calculations and estimations, as the needed capacity or its estimation is not known. However, thanks to the scale options presented in the proposed solutions, the only capacity limitation of the system are cost and available funds. Therefore, this limitation of missing cost evaluation does not hinder the validity of the results. Instead, it could help define the overall capacity selection of the individual resources.

A missing set of requirements constrained the choice of certain resources. For example, due to the lack of exact requirements for front end, crawler, and partly clean storage, a recommended resource or set of recommended resources is provided guided by existing information of these components and standard implementation model of these component types. The reason for missing the exact requirements is the ongoing initial research in those fields. Therefore, the architecture might be adjusted in the future to reflect the outcomes of those researches.

7.2. Future Work

This section will present open points, recommendations for future work, and outcomes. It will be structured into sub-sections, where each sub-section will provide the future work in the respective area.

7.2.1. Architecture

In the architecture concept as a whole are no significant changes expected. The individual sections and concepts were proven to be working well together. Despite the successful evaluation, when the FactCheck++ services are finalized, a few adjustments might be needed in the final amount of Function Apps or in the clean storage area. Such changes are expected during the lifetime and possible expansion of the application's functionality.

Apart from natural progression expansion, with the current state of cloud technologies, there is no significant architecture change expected in the foreseen future. However, with the frequent updates in cloud technologies, the sudden need to update the architecture to keep up with the industry standard is not excluded.

7.2.2. The Azure

Specific Azure resources will need to be defined in the short term. The Crawler resource is still an open topic, and the final Azure resource needs to be added to the diagram. The same applies to the front-end section, where the proposed App Service might change to Static Web App, or any additional needed resource might be added.

There is an expected change in other data storage resources regarding the clean storage section, apart from the XBase (CosmosDB Document NoSQL Database). Only the use of Document Database is certain and confirmed with multiple projects. The other data store resources presented in the clean storage section might be subject to change. The expected expansion and changes to the resources used should be only performed after research and based on the needs of the connected services. The use of data storage solutions from the Azure is not limited, as the Azure Data Factory can be connected with all solutions provided by Microsoft and many more. With this excellent flexibility for developers, the recommendation is to have either a responsible person for the architecture or add new resources only based on research and discussion to prevent unnecessary resources, keep the cost affordable and keep the big picture architecture principles intact.

The other Azure resources are set and should be used as set by the proposed architecture. The vital step before going live with the FactCheck++ is to configure the sizing and performance for the production system. The current prototype architecture is set to the lowest/cheapest available options for the development and test environments. These settings need to be updated as described in previous chapters. Despite the performance calculations, the best optimization can be achieved only based on telemetry data. Therefore, the telemetry data should be analyzed periodically to detect usage trends, bottlenecks of the system, or any other problems. The problems need to be solved preemptively before they become an issue.

Such proactive problem detection and management approach detects and fixes problems before they become an incident [21]. Such operation lowers the number of critical incidents and saves costs and reputation damages in the long term compared to reactive problem management [38]. A conclusion from [38] is that the best approach is to combine proactive and reactive problem management solutions. Therefore, the recommendation from this thesis is to not only focus on development but specify the problem management before

the application is productive. Regarding the Azure and architecture part, a proactive approach for detecting performance issues and keeping up with the newest cloud industry standards relevant for the FactCheck++ is crucial for staying up to date. The reactive approach mainly concerns software issues, but if any incident exploits an issue in the resource setup or architecture principle, it needs to be appropriately addressed.

7.2.3. Continuous Integration and Continuous Deployment (CI/CD)

CI/CD is the natural progression and should be addressed in the short to midterm future. CI/CD stands for Continuous Integration and Continuous Delivery. Simplified, such an integration environment provides the pipelines for building software packages (artifacts), running automated tests, and deploying the artifacts to the Azure.

It should consider both ARM template deployments and software ones. Such a pipeline setup for building and deploying all application software components will simplify the development and bring this project nearer to industry standards. A CI/CD environment was also requested in multiple feedbacks for the architecture, especially for the Azure implementation. Apart from software-related deployment, such pipelines should also be created for the ARM templates to simplify further the use and quick redeployment of the environment within Azure.

Looking in the long-term future, a dedicated Operation role should be considered. This role within the project should be responsible for creating and maintaining the Azure environment and its compliance with architecture, creating and maintaining CI/CD pipelines, monitoring the system, and supporting developers with the proper use of the Azure environment. Another possible responsibility is to monitor incidents and either solve them or assign them to the respective developer for solving. Creating such a role should be in long-term goals, as it will free the developers from system operation tasks.

7.2.4. Hybrid Cloud

Another mid to long-term recommendation from this research is to investigate other cloud providers and discuss if FactCheck++ could benefit from a hybrid cloud approach. Such hybrid solutions are becoming an industry standard for bigger companies deployed for important projects. The recommendation is to evaluate at least the following two approaches: (i) mixed runtime environment and (ii) backup system.

The mixed runtime environment approach means that the resources from different cloud providers would be used in the system. Such an approach can be used if some resources have better performance or pricing than the Azure ones. For example, an Amazon DynamoDB will be used for XBase instead of the CosmosDB, but the rest of the system stays in Azure.

The backup system approach duplicates the Azure environment within the other provider (e.g., AWS). This second system will be ready to use but only used if something happens to the primary system in Azure.

7.2.5. Summary

To sum up, the recommendations for future work, in the short term, the final decision for the crawler, front end, and other possible clean storage changes should be specified and added to the architecture diagram.

Next, in the midterm, the CI/CD pipelines should be defined for the ARM templates and software resources to simplify the development. Such initial changes will bring all the team members on track concerning unified environments and start forming the actual test environment where all finished implementation projects can be deployed and tested with other implemented components.

Going to the mid to long-term projects, the hybrid cloud research should be performed to determine the possible needs and, if found to be valid, provide the next steps in implementing such a hybrid approach. Also, the operation role responsible for maintenance, operation, and monitoring of the system should be discussed and investigated. Also, research on problem management should be undergone, where the recommendation is to focus on a combined approach of proactive and reactive problem management. The proactive approach should focus on the Azure runtime environment apart from software.

Lastly, through the lifespan of the FactCheck++ project, the architecture should be reevaluated with any significant cloud service updates, changes in the industry standards, and if any problems are detected based on system monitoring. Also, to ensure compliance with the architecture, the resource selection should comply with the reference architecture before adding any new resource to the system.

7.3. Conclusion

To conclude this research, it was shown that basing the architecture of the FactCheck++ is feasible. The research addressed all the needs and answered our research question if it is possible to design the reference architecture that would be cost-effective and performant for such a system. As the research has shown, using the public cloud approach is possible and recommended for start-up projects. The main benefit of using the public cloud, especially initially, is the financially affordable access to state-of-the-art resources without any upfront costs.

Based on the theoretical findings of plausibility, a reference architecture for the system was put together based on the FactCheck++ requirements. The final architecture proposal satisfies project and Azure requirements, bringing the compliant architecture which follows the recommendations and industry standards. The resulting architecture is also performant and cost-optimized.

As part of the proof of concept of the architecture was prototypical implementation, which also provides a practical evaluation of the proposal. Furthermore, the approach's usefulness can also be demonstrated by other projects that already used sections of the proposed architecture. Finally, the concepts are shown working with the actual implementation and test data.

To sum it up, based on feedback, the architecture proposal is a success and shows the

correct heading for the project. Furthermore, with the outlooks for the future discussed, the runtime environment can be further optimized, bringing the FactCheck++ architecture and environment to the level of any commercial, globally deployed web service.

Bibliography

[1] Thomas Boillat and Christine Legner. 2013. From on-premise software to cloud services: the impact of cloud computing on enterprise software vendors' business models. *Journal of Theoretical and Applied Electronic Commerce Research*, 8, 3, (Dec. 2013), 39–58. Number: 3 Publisher: Multidisciplinary Digital Publishing Institute. DOI: 10.4067/S0718-18762013000300004.

[2] James Carr. *Cloud Adoption Decision Framework for Startup and Small Organizations*. M.A. The College of St. Scholastica, United States – Minnesota. 76 pp. Retrieved 30th Jan. 2022 from https://www.proquest.com/docview/1929640908/abstract/C8858030B91F478EPQ/1. ISBN: 9780355080438.

[3] cephalin. App service plans - azure app service. Retrieved 22nd Aug. 2021 from https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans.

[4] craigshoemaker. Azure functions overview. Retrieved 28th Aug. 2021 from https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[5] craigshoemaker. What is azure static web apps? Retrieved 14th Aug. 2021 from https://docs.microsoft.com/en-us/azure/static-web-apps/overview.

[6] Cyberarc. What is least privilege access? PoLP definition. Retrieved 13th Aug. 2021 from https://www.cyberark.com/what-is/least-privilege/.

[7] dcstwh. Introduction to azure data factory - azure data factory. Retrieved 21st Aug. 2021 from https://docs.microsoft.com/en-us/azure/data-factory/introduction.

[8] dsk-2015. Data store decision tree - azure application architecture guide. Retrieved 21st Feb. 2022 from https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-decision-tree.

[9] dsk-2015. Understand data store models - azure application architecture guide. Retrieved 21st Feb. 2022 from https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-overview.

[10] EdPrice-MSFT. Medical data storage solutions - azure solution ideas. Retrieved 21st Feb. 2022 from https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/medical-data-storage.

[11] DB Engines. DB-engines - knowledge base of relational and NoSQL database management systems. Retrieved 21st Aug. 2021 from https://db-engines.com/en/.

[12] Hewlett Packard Enterprise. What is a reference architecture? – enterprise IT definitions. Retrieved 21st Apr. 2021 from `https://www.hpe.com/us/en/what-is/reference-architecture.html`.

[13] Cameron Fisher. 2018. Cloud versus on-premise computing. *American Journal of Industrial and Business Management*, 08, 9, (30th Sept. 2018), 1991. Number: 09 Publisher: Scientific Research Publishing. DOI: `10.4236/ajibm.2018.89133`.

[14] Moneer Helu, Timothy Sprock, Daniel Hartenstine, Rishabh Venketesh and William Sobel. 2020. Scalable data pipeline architecture to support the industrial internet of things. *CIRP Annals*, 69, 1, 385–388. DOI: `https://doi.org/10.1016/j.cirp.2020.04.006`.

[15] Jakob Hirschl. 2022. *FactChecking - Towards conflict analytics (dashboard) for content providers*. Master's thesis. Universität Wien, Vienna - Austria.

[16] JagVeerappan. Use azure spot virtual machines - azure virtual machines. Retrieved 22nd Aug. 2021 from `https://docs.microsoft.com/en-us/azure/virtual-machines/spot-vms`.

[17] Wolfang Klas and Peter Kalchgruber. 2017. Factcheck - identify and fix conflicting data on the web [vision]. Private Communications in April 2021. Vienna, (2017).

[18] Wolfang Klas and Peter Kalchgruber. 2020. Factcheck - introduction slides. Vienna, (2020).

[19] [SW], loud Native Computing Foundation Policy Repo, 12th Aug. 2021. URL: `https://github.com/cncf/foundation/blob/0556853d54e6d137e6a1eb6ae14bee7c20d66d68/charter.md`Retrieved 13th Aug. 2021 from.

[20] lpassig. Cloud migration antipatterns - cloud adoption framework. Retrieved 16th June 2022 from `https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/antipatterns/migrate-antipatterns`.

[21] Joseph Mathenge. Reactive vs proactive problem management. BMC Blogs. Retrieved 24th Apr. 2022 from `https://www.bmc.com/blogs/reactive-vs-proactive-problem-management/`.

[22] memildin. What is azure security center? Retrieved 21st Aug. 2021 from `https://docs.microsoft.com/en-us/azure/security-center/security-center-introduction`.

[23] Microsoft. AZ-204T00-A - Developing Solutions for Microsoft Azure (AZ-204 Microsoft Azure course/training material). Training material distributed to the participants of the AZ-204 Instructor led training course through skillpipe.com. (). Retrieved 22nd Aug. 2021 from `https://www.skillpipe.com/#/reader/urn:uuid:88438492-2a00-5769-bee1-e4c9ebc889fb@2021-06-29T03:14:03Z/content`.

[24] Microsoft. What is IaaS? infrastructure as a service | Microsoft Azure. Retrieved 28th Apr. 2021 from `https://azure.microsoft.com/en-us/overview/what-is-iaas/`.

[25] Microsoft. What is PaaS? platform as a service | microsoft azure. Retrieved 28th Apr. 2021 from https://azure.microsoft.com/en-us/overview/what-is-paas/.

[26] Dibran Mulder. 2019. Building a web scraper in azure. Dibran's Blog. (15th Feb. 2019). Retrieved 21st Aug. 2021 from https://dibranmulder.github.io/2019/02/15/Building-a-Web-Scraper-in-Azure/index.html.

[27] mumian. Bicep language for deploying azure resources - azure resource manager. Retrieved 3rd Jan. 2022 from https://docs.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview.

[28] mumian. Deploy resources with azure portal - azure resource manager. Retrieved 29th Nov. 2021 from https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-portal.

[29] mumian. Link templates for deployment - azure resource manager. Retrieved 12th Dec. 2021 from https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/linked-templates.

[30] mumian. Templates overview - azure resource manager. Retrieved 5th Sept. 2021 from https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/overview.

[31] neilpeterson. Browse azure architectures - azure architecture center. Retrieved 30th Jan. 2022 from https://docs.microsoft.com/en-us/azure/architecture/browse/.

[32] normesta. Azure data lake storage gen2 introduction. Retrieved 1st Sept. 2021 from https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction.

[33] Collaborators/Students of the FactCheck project. 2021. Azure guide. Internal Azure Guide distributed between Project Members as shared material. (2021).

[34] robvet. Candidate apps for cloud native. Retrieved 12th Feb. 2022 from https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/candidate-apps.

[35] robvet. What is cloud native? Retrieved 12th Feb. 2022 from https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.

[36] rolyon. What is azure role-based access control (azure RBAC)? Retrieved 12th Dec. 2021 from https://docs.microsoft.com/en-us/azure/role-based-access-control/overview.

[37] sakash279. Azure table storage support in azure cosmos DB. Retrieved 4th Sept. 2021 from https://docs.microsoft.com/en-us/azure/cosmos-db/table/table-support.

[38] Suad Sejdovic, Yvonne Hegenbarth, Gerald H. Ristow and Roland Schmidt. 2016. Proactive disruption management system: how not to be surprised by upcoming situations. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems* (DEBS '16). Association for Computing Machinery, Irvine, California, 281–288. ISBN: 9781450340212. DOI: 10.1145/2933267.2933271.

[39] tamram. About blob (object) storage - azure storage. Retrieved 15th Aug. 2021 from https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-overview.

[40] tamram. Introduction to table storage - object storage in azure. Retrieved 4th Sept. 2021 from https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-overview.

[41] Mario Villamizar et al. 2016. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (CCGRID '16). IEEE Press, Cartagena, Columbia, 179–182. ISBN: 978-1-5090-2452-0. DOI: 10.1109/CCGrid.2016.37.

[42] vladvino. Azure API management overview and key concepts. Retrieved 29th Aug. 2021 from https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts.

[43] Wenying Zeng, Yuelong Zhao, Kairi Ou and Wei Song. 2009. Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences Information Technology, Culture and Human - ICIS '09*. the 2nd International Conference. ACM Press, Seoul, Korea, 1044–1048. ISBN: 978-1-60558-710-3. DOI: 10.1145/1655925.1656114.

A. Appendix

A.1. Source Control

The source code used in the thesis, figures, and diagrams are present in the GitLab repository created for this thesis. This repository is located at Git01Lab and can be accessed with the following link: `https://git01lab.cs.univie.ac.at/thesis/2021ss/01638811-roman-szabo` or by scanning the QR code in the Figure A.1.

Figure A.1.: Master Thesis GitLab Repository URL Encoded as QR code

The ARM template source code with the short tutorial and "Deploy to Azure" buttons can be found in a dedicated repository under the FactCheck++ GitLab project. This repository is also located at Git01Lab under: `https://git01lab.cs.univie.ac.at/factcheck/arm-temlates` or by scanning QR code from the Figure A.2.

Figure A.2.: ARM Templates GitLab Repository URL Encoded as QR code

A.2. Azure Guides

Reference step-by-step guides for Azure are provided in following sections. These guides show how to create Azure resources, deploy template and perform after deployment actions. For beginners who did not work with Azure before, is recommended to take introductory learning path from Microsoft Learn. The path can be found under `https://docs.microsoft.com/en-us/learn/paths/az-900-describe-cloud-concepts/`.

A.2.1. Azure Portal Overview

The Azure Portal interface is simple to navigate. As can be seen in Figure A.3, the initial screen is split into 2 main parts, a Quick Access Bar and Recent Resources. The Quick Access Bar provides options to go directly to Azure Marketplace or to the overview screen of the last used resources. The Recent Resources, on the other hand, provide the list of last accessed deployed resources. This setup is easy to navigate and provide option to quickly access needed resource.

The navigation bar on the top is shared across all screens. The main part is the Global Search. This can be used to find deployed resource, Azure resource from Marketplace, Setting and much more. On the right is a collection of multiple icons. From these icons, the Azure CLI/PowerShell is the most used one. This button opens the Cloud Shell, an Azure CLI or PowerShell running in the cloud. It can be used to quickly manage resources.

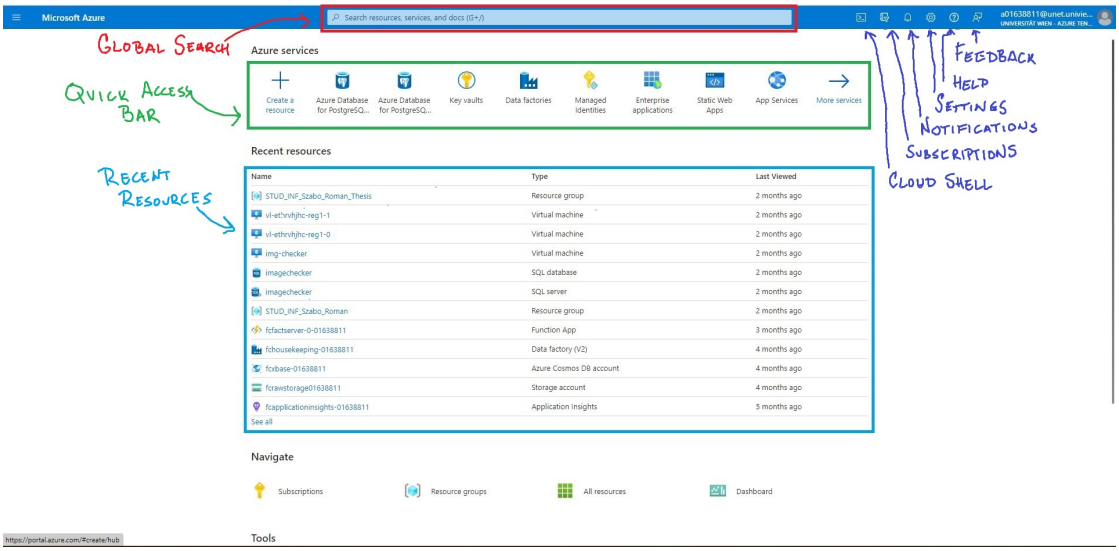


Figure A.3.: Annotated Initial Screen of the Azure Portal

The individual resources are using the same design principles as shown in example Figure A.4. The screen contains a Resource Menu for accessing resource settings, a Command Bar containing basic resource functions such as start, stop, and restart. Furthermore, a

resource details are shown followed by basic metrics overview or other useful information about the resource.

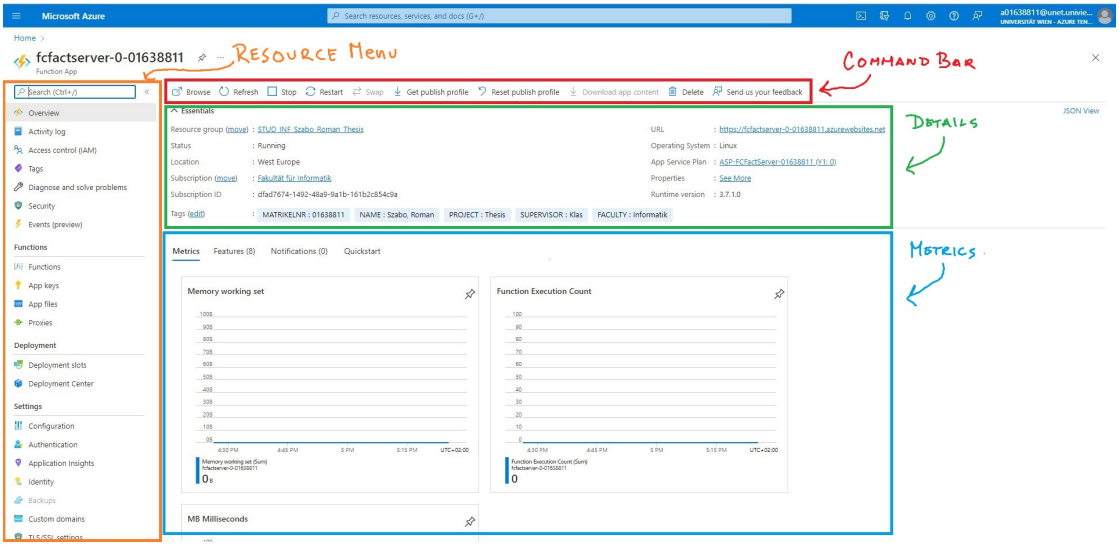


Figure A.4.: Annotated Resource Screen of the Azure Portal

A.2.2. Create Azure Resource

In this section will be shown the two methods on how to create a resource - Azure Portal and ARM Template. Further, options can be found in Azure documentation[1].

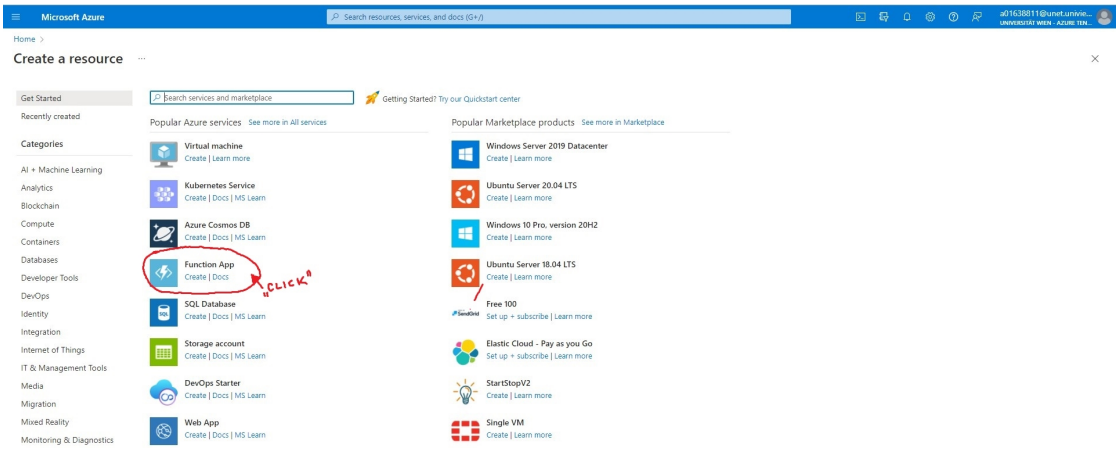
Azure Portal Method

Firstly, navigate to the Azure Marketplace, e.g. by clicking on "Create a New Resource" button in the initial screen. Then find desired resource in Azure Marketplace. Here, Azure Function are selected.

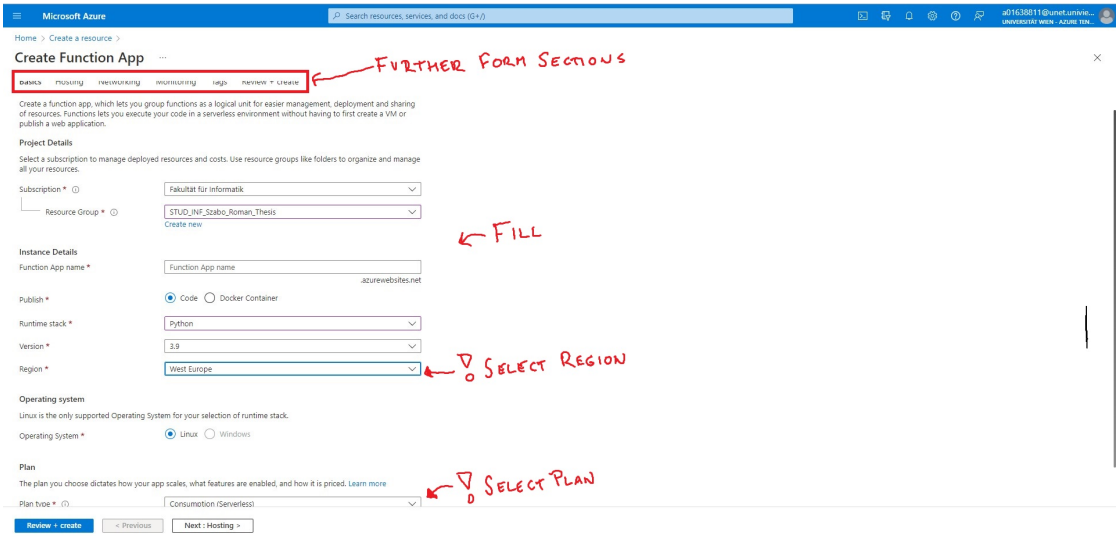
To create a resource, the create form needs to be filled. A resource group needs to be chosen, name of the resource, location (currently all FactCheck++ resources should be deployed in "West Europe"), consumption plan (for development always choose cheapest or serverless), and other resource-specific settings. Tags are used in the project as well. The default set of tags is depicted in Listing 5.2. When finished, click on the "Review + Create" button to let validate the input values. Once validated, the resource summary is presented and the resource will be deployed after clicking on the "Create" button. An example reference screen in available in Figure A.5

[1] https://docs.microsoft.com/en-us/

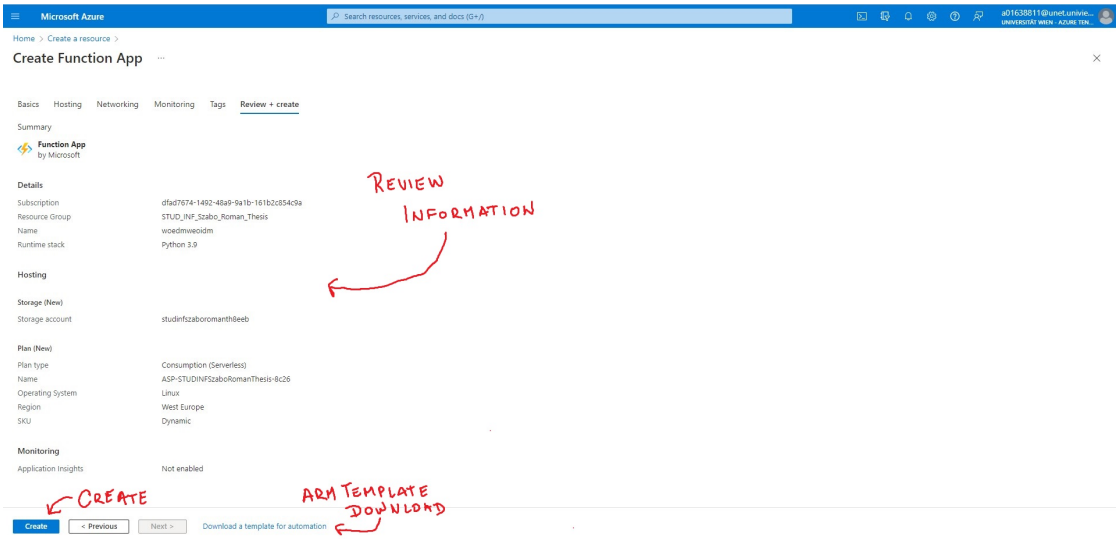
A. Appendix



(a) Azure Marketplace Screen



(b) Create Resource Screen



(c) Review and Create Screen

Figure A.5.: Create Azure Resource from Azure Marketplace

ARM Template Deployment

The resource can also be created by deploying one of the FactCheck++ ARM Templates. The templates can be deployed by clicking on the corresponding "Deploy to Azure" button, which can be found in the GitLab repository. Once clicked, a similar screen to the create screen will appear, as shown in Figure A.6. Here any required fields needs to be filled up (marked with *) and check/change any other parameters if default value is not correct/satisfactory for current needs. Please adjust Tags Array parameter by filling correct values: matriculation number, student name, project name, supervisor name, faculty. The default value represents object structure. With default value, the deployment may fail! A deployment form example is shown in Figure A.6

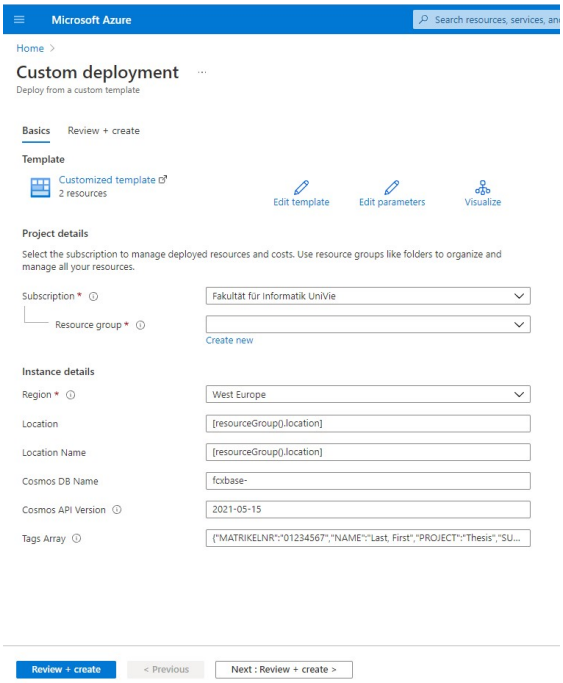


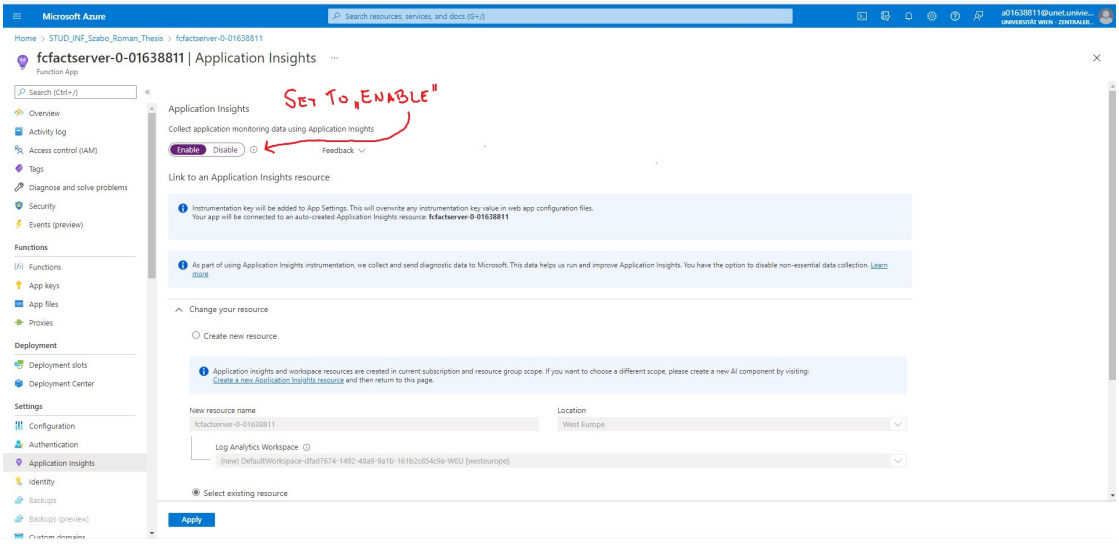
Figure A.6.: Template Deployment Example

A.2.3. Linking Application Insights to the Resource

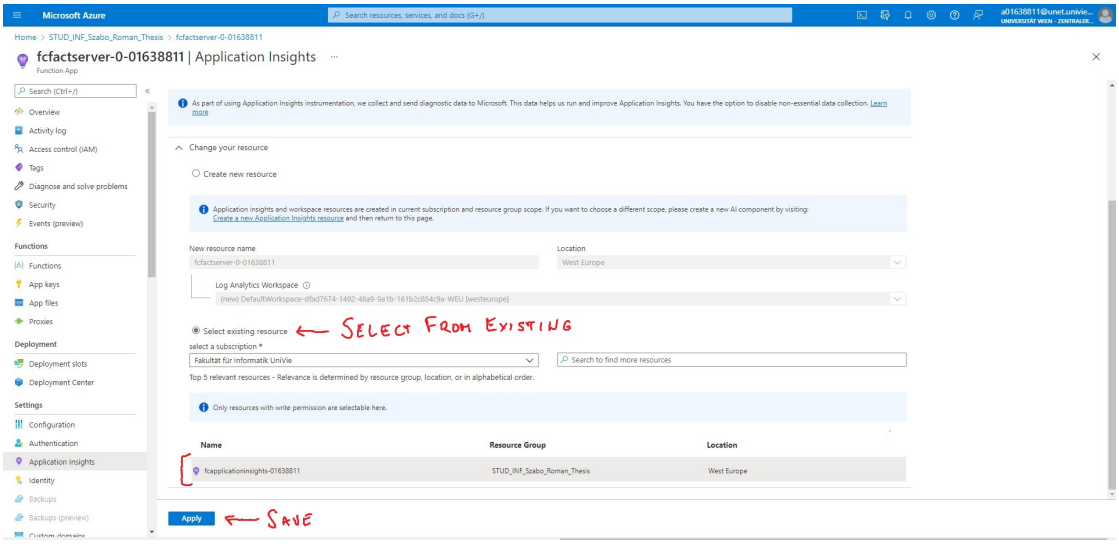
This guide will show how to set up Application Insights link to the Azure Functions. This is a post deployment manual task after deploying FactServer with ARM Templates. However, this steps apply for any Azure resource which can be connected to the Application Insights.

In the Resource Menu, Application Insights item needs to be opened as shown in Figure A.7 and clicked on the turn on button. Then, the slider needs to be set to the "Enable" position and the existing Application Resource needs to be selected. Then save the changes and the resource is linked to the Application Insights.

(a) Turn On Application Insights



(b) Enable the Application Insights



(c) Choose the Existing Resource

Figure A.7.: Application Insights Link Set-Up

A.3. "One-Page" Reference Guide

Short reference guide on how to choose Azure service based on the needed purpose. This reference guide is represented also graphically in Figure A.8. The order in which are the resources depicted corresponds to the preference.

Following, the Figure A.8 will be represented in the textual form with additional comments. The numbered list represents the preference, similar as for the figure self.

Front-end

- **Static Web Content**: an example, a web page - static HTML documents or scripts, can also be compiled output of the JavaScript frameworks as Angular or ReactJS
    1. Static Web Apps - ideal for web page hosting, it has bundled Azure Functions, which can be used for simple Back End needs
- **Other Web Content**: non-static web app or any software that requires compute (CPU and memory)
    1. App Services - ideal for server deployment, supports multiple languages in managed mode or docker containers, suited for web apps with server functionality

Middleware (Core Functionality)

- **On-Demand Execution**: execution that is triggered from an external source, such as HTTP call, event, or trigger, these triggered functions usually represents the business logic of the application and together forms an API
    1. Function App - ideal for event-driven scenarios, Azure Functions should be primarily used for FactServer or any triggered functionality
    2. Server Instance - ideal for monolith servers, or microservices that need to be running 24/7 or when Function App cannot be used due to any limitations, one of the appropriate resource (App Services, Spring Cloud, or containerized solution) should be chosen
- **Scheduled Execution**
    1. Azure Data Factory - ideal for large data movements or transformation at scheduled intervals, for any data related scheduled job, ADF (Housekeeping) resource should be used
    2. Function App - ideal for any lightweight job (e.g., status checking by pulling method), trigger can be a chron expression
    3. Logic Apps - ideal for fast created workflows that do not require fast execution (e.g., email sender or event trigger)

- **Process/Workflow Execution**
  1. Function App - in workflow setup, each function represents an execution step, functions can be linked to next execution step, by decisions, they can be forked or joined
  2. Logic Apps - contains many connectors to other resources and services, looping functionality, and logical operators
  3. Azure Data Factory - flow can be defined in the pipeline, ideal for big data processing flows

Back-end

- **Relational Storage**
  1. Azure SQL Databases - Azure managed SQL server of choice, ideal for any relational needs, allows for scaling, regional replication, and more (currently supported SQL databases are: PostgreSQL, Azure SQL Server, MySQL, MariaDB)

- **Key-Value Storage**: table like storage without any schema, the values are retrieved by its key
  1. Table Storage - ideal for key-value scenario, it is part of the Storage Account, indexed by partition and rows, which allows for fast lookups
  2. Cosmos DB (*Table API*) - basically same as Table Storage, but suited for high-demand scenarios, as it is more performant

- **Document Storage**: a typical NoSQL solution, is used as main storage type in the FactCheck++, consist of JSON documents stored in containers
  1. Cosmos DB (*SQL API, MongoDB API*) - the NoSQL solution, allows for fast access, various replication and consistency methods, ideal for current and commonly accessed or queried data
  2. Azure Data Lake Storage (*Containers, Files*) - ideal for documents accessed as whole file rather than querying, has access tiers for pricing optimization, has folder structure, ideal for big-data solutions
  3. Storage Account (*Containers, Files*) - similar as Azure Data Lake Storage, but for normal use

- **Other**: other storage needs of arbitrary file type, or other structures or queues
  1. ADSL or Storage Account (*Containers, Files, Queues, Tables*) - ideal for any needs, always choose the best storage type

Other Needs

- **Try PaaS Solutions**
  1. Azure Marketplace - for other needs, try to find the best resource in the Azure Marketplace, ideally it should be PaaS or SaaS solution
- **Fallback**
  1. Virtual Machine - as backup resource, if no Azure resource is suitable, Virtual Machine may be used

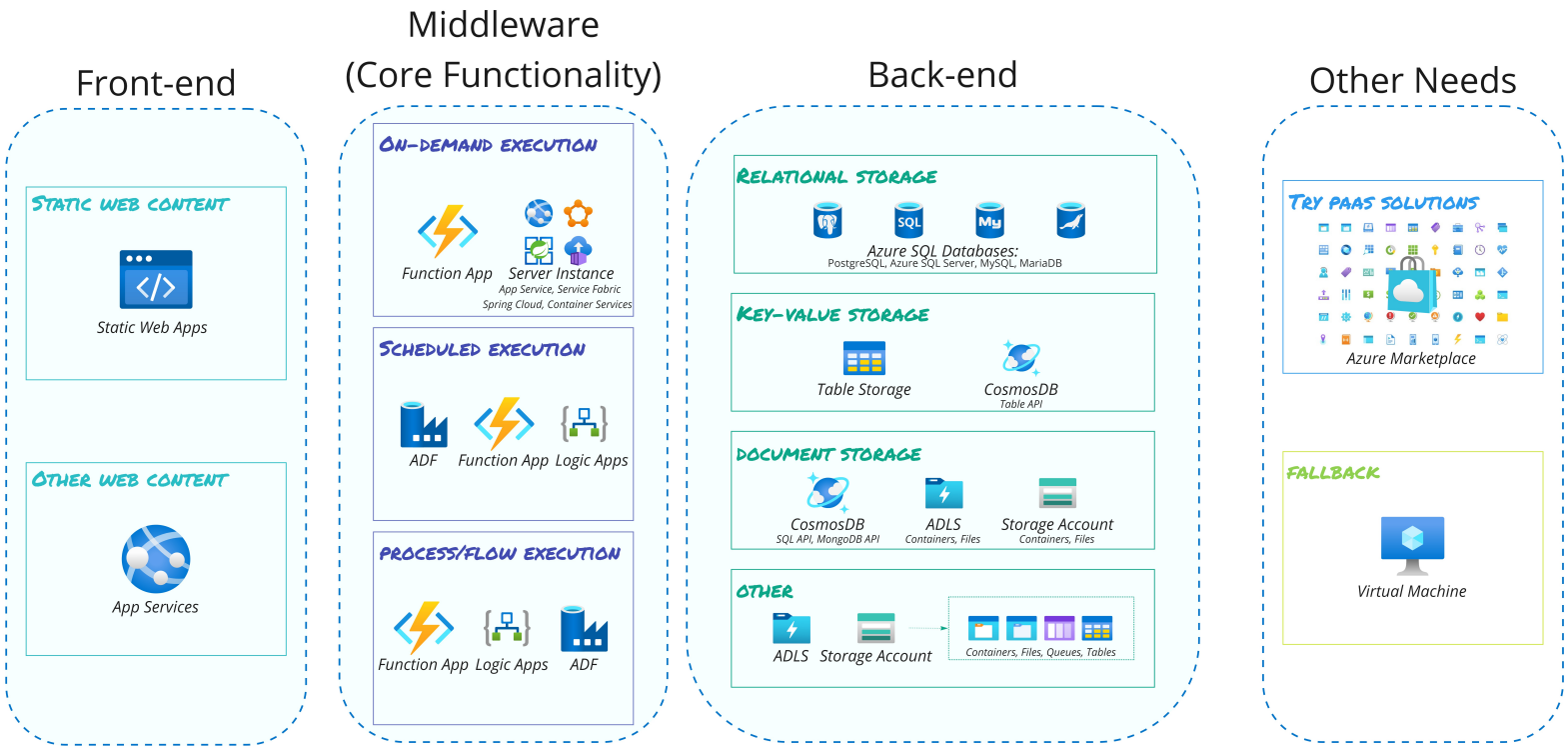


Figure A.8.: Illustration of the Reference Guide

A.4. Feedback Form

The original feedback form used for formal feedback collection can be seen at: `https://forms.office.com/r/x2uNbA4z9W` or scanning the QR code in the Figure A.9.

Figure A.9.: Form URL encoded as QR code

Feedback Questions

The feedback form consists of the following questions and possible answers:

1. How do you like deployment through ARM Templates? *Single choice, mandatory*
   - Star rating 1-5, where 5 is the best
2. How many times you have tried ARM template deployment? *Single choice, mandatory*
   - 1
   - 1-5
   - 5 or more
3. Did you encounter any errors? *Single choice, mandatory*
   - Yes
   - No
4. Please elaborate. *Textarea, only if previous is "No", mandatory if visible*
5. Which method of deployment did you tired? *Single choice, mandatory*
   - "Deploy to Azure" button
   - Azure CLI
   - Others
6. Please specify. *Text field, only if previous "Other", mandatory if visible*
7. Which template did you tried? *Multiple choice, mandatory*
   - Main Template
   - Dashboarding Web App
   - Middleware Main

- FactServer
- Application Insights
- API Management
- Data Store Main
- Raw Storage
- X Base
- ADF Housekeeping

8. How likely would you recommend use of the ARM templates over manual deployment from Azure Marketplace *Single Choice through Net Promoter Score®, mandatory*
   - 0-10, where 0 - unlikely, 10 - very likely
9. Is there anything missing from the Templates? *Textarea*
10. Any other feedback? *Textarea*

Feedback Form Answers

Two anonymous feedbacks were received through this form with following answers:
Responder ID 1:

1. 5
2. 1-5
3. Yes
4. "The factserver templates contains parameters which are only revealed in "Edit template" mode. If not modified there, the deployment results in an AuthorizationFailed error, because it points to a storage account outside my resource group."
5. "Deploy to Azure" Button
6. *N/A*
7. Dashboarding Web App, FactServer
8. 8
9. *Not provided*
10. "Using Azure Buttons feels like one the most compact solutions for sharing templated resources. The insights into Azure buttons will influence how I will distribute Azure resources in the future."

Responder ID 2:

1. 4
2. 1-5
3. Yes
4. "Building the middleware failed. It seems I have unsufficient permissions:"

```
{
    "status": "Failed",
    "error": {
        "code": "AuthorizationFailed",
        "message": "The client 'axxxxxxxx@univie.ac.at' with
    object id '6eb23316-4b17-46cd-94ab-c6879a5499c7' does not
    have authorization to perform action 'Microsoft.Storage/
    storageAccounts/listKeys/action' over scope '/
    subscriptions/dfad7674-1492-48a9-9a1b-161b2c854c9a/
    resourcegroups/STUD_INF_Szabo_Roman_Thesis/providers/
    Microsoft.Storage/storageAccounts/fcrawstorage01209758' or
     the scope is invalid. If access was recently granted,
    please refresh your credentials."
    }
}
```

5. "Deploy to Azure" Button
6. *N/A*
7. Main Template, Middleware Main
8. 10
9. *Not provided*
10. "Deployment takes pretty long but I guess there is just a lot happening. I would really like to have some more details what to do with the component once they are deployed. Unfortunately I did not get the main template running but even if I would be a little lost what to do next."

Note: The Error mentioned in both responses was fixed, the feedback was not re-submitted by responders afterwards.