Introduction

In the recent years, we can see a significant increase of trading volumes on the most liquid market, the US stock market, generated by retail investors[1]. There are multiple definitions for a retail investor, according to the Campbridge Dictionary retail investors are:

*" A member of the public who makes investments, not a large organization or business that makes investments." [3]*

I am going to expand this definition with the following:

"Retail investors are non-professional members of the public who make investments", with emphasis on the non-professional part.

Numerous studies[4][5][6] has looked into the overall performance of this type of investors and found that their vast majority are losing on the long term. Not suprisingly, considering their characteristics which was described as: inexperienced, aggressive in terms of trading, speculative, their outperformance would be expected by sheer luck, and overconfident.

This segment are also more likely to engage in trading activity, on the basis of non-fundamental information. During the dotcom bubble sometimes they responded to important news regarding to certain companies by quickly buying similarly named, but distinct companies. Such events are still not uncommon, it has happened recently when a misinterpreted social media posts caused a company value skyrocketing[1]. No wonder that the financial media coined this group as "dumb money", in comparison with the big institutional investors and mutual fund companies, named as "smart money".[7]

Their composition however, has changed over time. While in the late 1990s the mean retail investor was 50 years old and invested around $47,000, nowdays, they are younger, around 31, and has less to invest, between $1,000 and $5,000.

With the crowd, the platform and communication landscape has changed as well. While in the 1990s, the online chatrooms, newsletters and forums were the main ways of communication, and slow internet or expensive telephone brokerages were the common tools of trading, these days the social media platforms serves as the main way of communication, and the newly surfaced trading or financial technology (fintech) firms, such as Robinhood, Revolut, or Wise are the ones who are channeling the huge amount of retail volumes into the market[2].

Yet the question arises, if retail investing hold such risks and underperformance, why are it on the rise?

Part of the answer lies in the new platforms, accompanied with the herd like behaviour generated by the fast information exchange on social media sites, however I rather intend to focus on the platforms now.

The 2010s brought huge technical innovations. In this decade smartphones and fast mobile internet speed became common in the developed world, which has also influenced -among others- the financial services industry. With the advent of fast and constantly innovating mobile operating systems it is also served as the breeding ground for new mobile application development technologies that allowed to create services that previously was existed only in brick and mortar form. Nowdays, we can open a bank account in some countries, without ever stepping into the bank's building, if such building exists at all, besides the development offices of the fintech company that has acquired a banking licence.

The same change has happened with brokerages. Fintech companies has challenged the "old world" brokerage structures, the ones that stopped innovating in the IT or weren't fast enough, have risked their own existence. The new challengers brought everything that this new generation of retail investors needed: With the new technologies they made access to the stock market as easy as never before, the registration process was quick, interactive and fun as new IT specialisations emerged such as UX Design, that made the user engaged with the application as much as possible. The easy to use interface design, the commission-free execution -that forced the whole brokerage market to follow- became possbile with the established new alternative income streams such as selling customer data, providing order flow/liquidity to exchanges for incentives. Even provided out of reach shares to the public via the opportunity to trade fractional shares, as well as simplifying options trading, which was due to it's complex and abstract derivative nature, previously utilised much less by this segment.

Given all of this new, and innovative features, no wonder that commercial success and user growth has followed:

*"A Deutsche Bank survey found that almost half of US retail investors were completely new to the markets in the past year. They are young, mostly under 34. And they are aggressive: much more willing than those more experienced in stock markets to borrow to fund their bets, to make heavy use of options to fire up wagers on stocks, and to use social media as a research tool to find trading ideas."[2]*

Retail investors can now invest without all the fuss at the dormitory, on the tram, or over university lunch breaks on cleverly-designed smartphone apps. Easy access however, comes with appaling consequences:

*"Access to leverage — in the form of “margin” loans from brokerages or financial derivatives like options — is also freer than ever before. US margin debt soared to a record $799bn in January. "[2]*

It seems that despite the losses, such features (accompanied with government stimulus in the US, from what -a recent survey responders tell- 37% will go into the markets [2]), are hard to beat, and the very recent history showed that the synergy between these participants are working quite well. Dumb money deserves it's name, after all.

The questions is, if we have an aspiration for investing in the long term on our own, yet does not want to participate in the flow of the "pump and dump" herd, what can we do about it?

Turns out several things, which I will explain in details during the later chapters, but all of these methods has the same foundation, namely a Market Analytics Platform (**MAP**), which is going to be the topic of this thesis. As a Software as a Service, it helps us to generate alpha over the long term and hopefully raise us above the bar of dumb money.

I will use a bottom-up approach in the development and consequently in my thesis.

First, I will introduce the architecture and development methodology, followed by a short presentation of the used technologies. I will dive into the explanation of the software's planning, estimation, and development after that, ending it with the UI design description.

In the middle of the thesis, I attempt to ellaborate on the alternative solutions regarding to the technologies on the Backend, Frontend, and DevOps side. After that, Ill get into some details about the market analytics toolset that this program will contain, along with the introduction of other business processes, for example report generation.

At the end, I am going to conclude my findings, and present a usecase for the completed software.

Development Methodology

Among the very first steps of an application design, there is the need to choose a Software Development Methodology (SDM) framework, as choosing a proper methodolgy can increase the project success rate [8]. There are several frameworks, methods, to be choosen from, and it very much depends on the project itself, which framework would be best to use, or worst.

A development methodology helps the team working on a project by providing a collection of processes, techniques, and toolkit that helps, organises, and standardise the processes throughout the project lifecycle [8]. Such methods can be related to workflow management like Scrum, or can be a directly applicable development procedure, like the Test Driven Development (TDD).

As I mentioned earlier, there are several frameworks, methods, to be choosen from. To be able to see the complete picture, I am providing the following image:



Image I. source: self-edited illustration

SDMs are all models that exist to support the Software Development Lifecycle (SDLC), while SDLC itself is a subset of the System Development Lifecycle.[9] SDLC can be described as:

"*... process of building or maintaining software systems[7]. Typically, it includes various phases from preliminary development analysis to post-development software testing and evaluation. It also consists of the models and methodologies that development teams use to develop the software systems, which the methodologies form the framework for planning and controlling the entire development process."* [10]

There are several SDLC models [9][11], Image I. shows the following:

* Rapid Application Development (RAD)

The RAD method enables information systems to rapidly transition from design phase to completion, while keeping costs relatively low. The system is split up to smaller packages, which helps to make changes while the project is still in development. For each package, there are defined deadlines for delivery, that should not be exceeded. The project's requirements can be reduced to match the deadlines.[8]

* Waterfall

In Waterfall modell, the software is split up into multiple, consecutive parts, with some overlap is tolerable between parts. The focus is on design, schedules, deadlines, budgets, along with the overall development of the whole software at the same time. WIth extensive documentation process, reviews, and approval from the customer and the managment are all needed at the end of most parts, therefore there is strict governance built into this model.[11]

* Agile

Agile comprises of practices and frameworks/methods that include requirements discovery and solutions improvement through the collaborative effort of self-organizing and cross-functional teams with their customer(s)/end user(s), adaptive planning, evolutionary development, early delivery, continual improvement, and flexible responses to changes in requirements, capacity, and understanding of the problems to be solved.[12] [13]

Based on my previous work experience, I have choosen Agile as my preferred SDLC model. As shown, Agile consists of Frameworks/Methods and Practices.[14] [15]

The following are displayed on Image I:

**Methods:**

* Extreme Programming

Extreme Programming (XP) is an agile software development model with the goal to improve the software's quality and it's developers life quality. In it's core, it defines interconnected engineering practices for software development. These practices are advised to be done in conjunction as they can reinforce each other.

Extreme Programming practices are like: The Planning Game, Small Releases, Metaphor, Simple Design Testing Refactoring Pair Programming Collective Ownership Continuous Integration 40-hour week On-site Customer Coding Standard. [20]

* Scrum

Scrum is a model used to handle software development projects and other knowledge work.

Scrum is practical in that it provides a frame for the development team to set up their own theory of how something works, test it, and then make a suitable adaptation according to the experience.

Due to Scrum's structure, it allows other practices from different frameworks to be used within it's context.[21]

**Practices:**

* Test-driven development

Test-driven development or TDD for short, is a style of programming where the following 3 actions are closely tied: Design (refactoring), Testing as unit tests, and Programming.

It can be summarised as: [22]

1. Create a unit test related to an aspect of the program

2. Run it, and see it fail, as that aspect has not yet been implemented.

3. Make the minimum amount of code, that would make the test pass.

4. Refactor the code until it contains no duplication, pass the tests, has clear boundaries between it's parts according to each part's responsibility, and uses the minimum required number of methods, classes, etc.

5. Repeat [22]

* Domain-driven design

Domain-Driven Design or DDD for short, is a software development style that focuses on creating a domain model that has

clear yet deep concepts about the rules, and methods within the domain.

A 2003 book by Eric Evans described the name of DDD, and it's approach with a list of patterns.

Since it has been developed further, and it works particularly well in complicated domains where a lot of logic needs to be organised.[23]

Based on my previous work experience, I have choosen to base the application design on the Domain-driven design principles, while applying the Test-driven development processes during development. The former will provide a clear connection between the business domain and technical concepts, and the latter will ensure the code quality.

Architecture

By deciding an architecture, we are defining the skeleton of the application. It will set the boundaries, and structure of the development. No complex application can be built without a clear architecture, just like no house can be built without a clear blueprint.

There is a vast number of architectural patterns to choose from.[16] The choice depends on the type of the application in development [16], and the type should be based on business understanding and the software's usecases.

Here are a few application types and their related architectural patterns:



Image II. source: self-edited illustration

Distributed System type:

Example: web applications, email

* Client-server

Client/server architectural style details a distributed systems that require an independent client and server, with a network between them. The most basic form of client/server architecture imply a software on the server, that is reached directly by numerous clients. It is often called as a 2-Tier architecture. [16]

* Service-oriented

In Service-oriented architectured applications, the functionality is delivered as a set of services, and the architecture allows the creation of applications that make use of software services. The services are loosely coupled as they use interfaces that can be called, published, and discovered. [16]

Structured type:

Example: Pluggable applications, UNIX, TCP/IP protocol suite

* Layered

When in an application architecture we focus on grouping related functionality into separate layers that are built on top of each other, we have a so called Layered architecture. Each layer's logic has a common role/responsibility. The layers communicate with each other in a loosely coupled manner. The layers helps to archive a strong separation of concerns while prividing maintainability and flexibility. [16]

* Component based

In Systems Design, the Component-based architecture is describing a software engineering approach. With well designed interfaces containing functions, variables, it comprises of separate logical components. The emphasis is on archiving a higher level abstraction (compared to OO design) without spending too much time on state management and communication protocol issues. [16]

The recent trend of the past few decades is a shift towards modularization, loose coupling, and distribution. [17] This trend became more prevalent, as digitalization appeared throughout various industries and with that, there is an increasing need towards software quality and maintainability. [17] While softwares are growing more and more complex, reaching enormous amount of users, one step toward this trend is the use of Service-oriented Architecture (SOA). [17]

There are multiple subsets, approaches to a Service-oriented Architecture [18], here I am going to talk about one of them, that is going to be the choice of my architecture: Microservices.

Microservices

In the literature, there are several definitions available for Microservices:

* “*A microservice is a cohesive, independent process interacting via messages.*” [18]
* “*A microservice architecture is a distributed application, where all its modules are microservices.”* [18]
* *"Loosely coupled service in a bounded context " [18]*
* *"Small autonomous services that work together, modelled around a business domain."* [18]
* "*A small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility.*" [18]

Among these I am going to apply the last definiton, when designing the architecture in my thesis. It contains all the -few, but important- basic principles of a microservice architecture:

* Bounded Context

Related functions are combined into a single business ability, and each microservice implements one such ability. In this way there is a perfect symmetry between business abilities and system composition, making it easy, for example to know where a function is, in order to update or fix it. [17]

* Size

The focus on small size is a key component of microservices compared to the previous SOAs. Idiomatic use of microservice architectures suggests that if a service is too large, it should be broken into two or more services, thus preserving granularity and maintaining focus on providing a single business ability only. The smaller size brings crucial benefits in terms of service maintainability and extendability: a small service can be simply changed, and if needed, rebuilt from the start with limited resources and in short time. [17]

* Independency

Microservice encourages loose coupling and high cohesion by expressing that every microservice in microservice architectures is operationally free from others, and the only form of communication between services is through their interfaces. This is foundational since this allows one to change, fix or upgrade a microservice without compromising the system correctness, provided that the interfaces are kept. [17]

In essence, microservices are small applications -independent softwares- that are interacting with each other.

Communication

When talking about communication in the Microservice context, first, we have to distinguish two types:

In-Process and Inter-Process communication.

Communication between different services across the network are called Inter-Process communication, while communication within a service is called In-Process communication, and they are very different in nature. Among these differences I would like to highlight three, namely performance, handling change and errors.

Performance

The main difference between the In-Process and Inter-Process communication performance is that while during In-Process calls, the compiler and runtime may be able to utilise several optimization techniques to make the call the most efficient, while in Inter-Process calls such optimizations are limited, as here packets have to be sent. An illustrative example would be the following:

In an In-Process method call, when I pass a parameter to this method, this data structure doesn't move anywhere, as I am likely to pass just a pointer in a memory location.

In an Inter-Process call like making calls among microservices on the network, the previously mentioned data structure would needed to be serialized that can be transmitted, and on the receiver side, it is needed to be deserialized, both which takes time. Therefore we are very much need to think about the payload size.

Compared to In-Process communication, utilising services that communicates in such a distributed environment in a demanding manner -meaning making several request to each other across the network- are needed to be carefully created with efficient serialization mechanism and reduced amount of data usage. Even changing the whole data handling functionality might make sense performance wise. Like instead of taking the data as a parameter, it can also be offloaded into a filesystem for example.

Handling changes

Changing an interface inside a single process -normally- has no additional burden on the whole rollout procedure. The interface is packed together with the caller, and usually a modern IDE even support built-in refactoring abilities for changing method signatures. Such deployment can be done as one atomic step.

When such interface are used among several other independent services the solution is not so simple once it comes to changes. For example, a backward-incompatible change to a microservice interface would force us to use new techinques, such as lockstep deployment to handle the new contract rollout.

Handling errors

The nature of errors are usually different in a distributed and in a single process environment.

In the latter, the errors are likely to be deterministic. Either they are expected and can be easily handled, or disastrous, in case we just push the error up to the call stack.

Within a distributed environment there is a serious exposure of errors that are outside of our scope.

For example downstream service outages, network timeouts, container crashes due to excessive memory consumption.

In the book, Distributed Systems by Andrew Tanenbaum and Maarten Steen[25] distinguishes 5 types of failure modes within Inter-Process communication: Crash failure, Omission failure, Timing failure, Response failure, Arbitrary failure.

Below are the short descriptions of these modes:

Crash failure

Server is suddenly crashed.

Omission failure

Response is missing, or a downstream service for example stops firing events.

Timing failure

Something has happened too early or too late.

Response failure

Incorrect response recevied.

Arbitrary failure

Situation when something went wrong but the participants cannot agree if the failure has occurred or why.

Some of these errors might go away easily, but some might be persistent. Therefore in distributed systems we need a richer set of returned error descriptions, for example HTTP is a protocol that well understands this requirement.

Inter-process communication technology overview

There are several styles of inter-microservice communication exists, with different technologies:



Image III. source: [24] page 94.

Synchronous blocking

The operations are blocked while waiting for a response.

Asynchronous nonblocking

The operations are continuing after the call has been made without waiting for a response.

Request-response

After the call, we are expecting the result in a response.

Event-driven

Events are emitted that might or might not be consumed by other services. The event emitter has no knowledge of the consumers.

Common data

A shared data source used among services.

There are several things to consider which style is preferable for a given project. Choices like: preferred technologies such as message brokers, or whether there are specific latency targets to archive, or the need for ability to scale, even security related aspects can influence the decision.

For the Market Analytics Platform project, I will utilise both synchronous and asynchronous communications.

For synchronous communication I will use REST over HTTP technology which is a request-response collaboration style, and for asynchronous communication I will use a topic-based message broker Apache Kafka, which can be used to support event-driven collaboration style.

Before I introduce the Market Analytics Platform project's architecture, I would like to write about

an implementation technique named Saga for business processes crossing through multiple services, and two common technologies associated with the microservice world, namely:

Service Mesh and API Gateway.

Service Mesh

Simply put, a service mesh holds the common functionalities of inter-service communication.

This approach comes with two benefits:

* Reduces the code needed for microservices to implement themselves.
* Consistency of common functionalites.

For example, service discovery, or load balancing are such functionalities that are common within a service mesh. Alternatively, these logics can be used within a library however, the disadvantage would be the increased maintenance time as each microservice must be updated with each new version of the library, as well as issues arising in the meantime when separate microservices are using different versions of the library.

A service mesh is utilising proxies (so-called sidecar proxy or data plane) next to each service. A single business process can trigger multiple inter-process (or "east-west" in data center lingo) calls. Such amount of calls can place a heavy load on the proxies, and service mesh implementations race to limit the impact while seamlessly providing the above stated functionalities. A control plane is also part of the service mesh, it acts as a place where the proxies are managed, where their behaviour can be changed, as well as collecting informations about them.

I will use a Service Mesh in my thesis work.

API Gateway

API Gateway focuses on handling requests coming from external sources into the microservice environment, in other words dealing with "north-south" traffic.

They mainly act as a reverse proxy, in addition to providing for example, logging services, rate limiting, or servicing API keys to external parties.

Projects where huge amount of external parties are expected to use our API directly, an example would be a data provider software or a headless CMS platform are what an API gateway is for.

Using an API Gateway for such softwares, that are not part of this "API Economy" (or have their own GUI clients to access their microservices) would be just an increase to their complexity and making them more volunerable and error prone by implementing this feature.

Besides some technical reasons (for example in case of Kubernetes, a simplified API Gateway implementation is needed as it cannot handle the communication to and from it's cluster by itself) there have to be a careful analysis before using it.

I have decided not to use API Gateway in my thesis work.

Distributed transactions

Implementing business logic that spreads through multiple services can be tricky by itself and even more in a distributed environment.

In such cases we would like to archive that all changes to occur as a single unit, meaning either all changes complete successfully or no changes occur at all. Most of the time this requirement point us towards the use of a database transaction.

In a database transaction we can be sure that part of the CRUD (specifically, the **c**reate, **u**pdate and **d**elete) operations are completed successfully, on multiple tables as well. The key properties for a database transaction can be summarized in the ACID acronym.

In short, ACID stands for:

Atomicity

Means that all operations within a transaction either all complete or fail, leaving no interim states in the database.

Consistency

All changes in the database keep the data's integrity, or it doesn't happen at all.

Isolation

Multiple transactions can run at the same time without any issue as each transaction's non-committed changes are not invisible for the others.

Durability

In case of a system failure we can be sure that committed data remains in the database, intact.

Business processes that occur within a single microservice environment involving only the microservice's own database (or if multiple microservices are working on the same database at once) can easily archive ACID properties.

The outlook is a bit different if the business process has to be implemented across microservices that each has it's own database. In this case, we have to split up an otherwise single transaction into two or multiple ones. In such multiple depending transactions treated as one single business operation, we have lost the atomicity property on the whole operation. One common solution that is trying to solve this problem is the use of a distributed transaction, more specifically a two-phase commit implementation.

Two-phase commit - 2PC

Two-phase commit is a distributed transaction algorithm that helps in case multiple separate processes needs to make transactional changes in a distributed system. A two-phase commit -as it's name says- consists of two phases:

* Voting phase
* Commit phase

There is a so called central coordinator mechanism involved which is handling these stages. In the voting phase, the central coordinator asks the workers, involved with the transaction, whether or not the state changes can be made. If all workers confirm that the state changes can occur, the commit phase comes next. If any worker says that there is a problem, for example a constraint violation, and the change cannot occur, then the entire operation aborts and no changes has been made.

The only way a worker can safely say that the change can occur in the database, is through the use of a locks. In a small scale operation, that involves a few tables and happening rarely, this cause no harm. However, in a more resource intensive scenario, meaning a business transaction that involves many tables and spans through several minutes, or even hours, days, or just simply occurring multiple times in a very short timeframe, it can cause vast amount of errors, that are hard to take care of.

For example, as there can be latency between the participants of the transaction, meaning part of the workers had already committed their transaction yet another part of the workers not, that means on the whole single business operation we have lost the ACID's isolation property. As workers need to acquire and hold locks until the second phase, it can lead to several issues regarding to lock management, and deadlocks. Problems related to workers, such as a worker is not responding to the commit request from the central coordinator can lead to troubles that needs to be handled manually by an engineer.

For these reasons, it is generally not advised to implement two phase commits, or use it just for rarely occuring, short living operations. There are some well implemented distributed transactional algorithm implementations that can be used successfully on large-scale databases, for example Google's Spanner. However for Google to succeed in this area it had to, for example, invest in very expensive datacenters and use satellite-based atomic clocks which on its own shows that to archive such a feat correctly is not a simple business.

However, if we still have the need to manage large business transactions spanning through multiple microservices and their databases, there is a better alternative: Sagas.

Saga

At its core, a Saga is no more than an algorithm that is coordinating multiple state changes while avoiding the need to lock resources for longer than needed periods of times. Designing a Saga involves modelling the whole business process as discrete steps that are executed one by one. Effectively we are creating a sequence of transactions. Each of these transactions by itself complies the ACID properties, as it happens within the respective microservice's database, while the Saga itself does not.

In case a failure happens, there are three common recovery modes:

* Backward recovery
* Forward recovery
* Mixed recovery

Backward recovery is basically the standard rollback function steps, such as reverting and cleaning up all the failed states. Here, we create compensating transactions that helps to revert previously committed processes. In case of a forward recovery, the algorithm start over at the point where the failure has occured, and keeps executing from there. As its name suggest, the third option involves both backward recovery and forward recovery processes, a valid scenario could be that after a certain tries of the forward recovery algorithm, if the an error keeps occurring it switches to the backward recovery algorithm to revert all previously committed states.

Important to point out that Saga can only recover from business failures and not technical issues. A business failure is for example at a merchant's system, the reserved stock's product id in the warehouse database does not exist in the accounting database. However, if the accounting service throws an Internal Server Error because the accounting service bean was not populated by the dependency injection framework due to missing default constructor signature in it's class, thats an issue that needs to be handled on its own, a Saga cannot recover from that. In that sense, the Saga is expecting to work with proper components, but in case we would like to handle the technical errors as well, there are some solutions that builds on a Saga, effectively creating a layer above it to handle such technical failures.

Saga implementations

There are two types of Saga implementations:

* Orchestrated Saga
* Choreographed Saga

Orchestrated Saga is using a central mechanism (== orchestrator) to conduct the order transactions and issue a compensating transaction if required. This central orchestrator can become an issue later on, as it introduces domain coupling as well as it might take on logic that otherwise should have been placed inside a service, basically centralizing the logic.

Choreographed Saga on the other hand tries to distribute the responsibility between the affected services, effectively crating a reactive environment. The events are circulating in such a system, and only affected services reacting to them, allowing more room for parallel processing and this way might make the business transaction even faster. Usually a message broker handling the events around the system for example RabbitMQ, or Apache Kafka. Due to the event-driven nature of this type of Saga implementation, the services doesnt know about each other, therefore the domain coupling is greatly reduced, compared to Orchestrated Saga implementation.

At the same time in such a decentralized system, it can be hard to know in case of an error what has happened and how was it happened. A simpler solution for that is the use of a unique correlation id for the Saga, that is passed around for that Saga's events. This way, with logging process and a specifically designed service with compensating transaction algorithms can then put together the business transaction, then detect and correct from the point where it has failed by issuing the appropriate reverting transaction, in case the originally involved services were not able to do it.

One way or another, in case of a choreographed Saga, the ability to trace the Saga transaction events is imperative.

habar immense litreature exists about the theory of microservices and related technology, now I intend to focus on the actual technical implementation of the software for two reasons:

the required upper boundary of pages of the thesis

the thery probably best can be eplaned along with the actual implementaion.

[1] S. Aramonte, F. Avalos (01 March 2021): The rising influence of retail investors https://www.bis.org/publ/qtrpdf/r\_qt2103v.htm (download date: 2021.08.30.)

[2] K. Martin, R. Wigglesworth (09 March 2021): Rise of the retail army: the amateur traders transforming markets https://www.ft.com/content/7a91e3ea-b9ec-4611-9a03-a8dd3b8bddb5 (download date: 2021.08.30.)

[3] Cambridge Dictionary https://dictionary.cambridge.org/dictionary/english/retail-investor (download date: 2021.08.30.)

[4] B. Barber, X. Huang, T. Odean, C. Schwarz (23 Oct 2020): Attention Induced Trading and Returns: Evidence from Robinhood Users https://papers.ssrn.com/sol3/papers.cfm?abstract\_id=3715077 page 28-29 (download date: 2021.09.01.)

[5] F. Chague, R. De-Losso, B. Giovannetti (22 Jul 2019): Day Trading for a Living? https://papers.ssrn.com/sol3/papers.cfm?abstract\_id=3423101 page 5 (download date: 2021.09.01.)

[6] B. Barber, Y. Lee, Y. Liu, T. Odean (May 2011): The Cross-Section of Speculator Skill Evidence from DayTrading http://faculty.haas.berkeley.edu/odean/papers/Day%20Traders/Day%20Trading%20Skill%20110523.pdf page 23 (download date: 2021.09.01.)

[7]A. Bell (19 March 2020): How Dumb Money Can Become Smart Money https://www.investopedia.com/investing/surprising-benefits-when-brokers-grade-their-customers/ (download date: 2021.09.01.)

[8] CV Geambaşu, I Jianu, I Jianu, A Gavrilă (2011): Influence factors for the choice of a software development methodology page 480,482,483 https://core.ac.uk/download/pdf/6261795.pdf (download date: 2021.11.25.)

[9] A Mishra, D Dubey (5 Oct 2013): A Comparative Study of Different Software Development Life Cycle Models in Different Scenarios https://www.researchgate.net/publication/289526047\_A\_Comparative\_Study\_of\_Different\_Software\_Development\_Life\_Cycle\_Models\_in\_Different\_Scenarios page 64 (download date: 2021.11.25.)

[10] YB Leau, WK Loo, WY Tham, SF Tan (Feb 2012): Software Development Life Cycle: AGILE vs Traditional Approaches https://www.researchgate.net/profile/Leau-Yu-Beng/publication/268334807\_Software\_Development\_Life\_Cycle\_AGILE\_vs\_Traditional\_Approaches/links/546989b40cf2f5eb1804f3d1/Software-Development-Life-Cycle-AGILE-vs-Traditional-Approaches.pdf page 162 (download date: 2021.11.25.)

[11] V Rastogi (2015): Software Development Life Cycle ModelsComparison, Consequences https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.667.9896&rep=rep1&type=pdf page 168 (download date: 2021.11.25.)

[12] K. Collier (2012). Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing https://accord.edu.so/web/content/69184?download=true&access\_token=a6c3a1a0-9efd-408f-a7d7-4a57967a3de0 page 3 (download date: 2021.11.25.)

[13] K Beck, M Beedle, A Van Bennekum, A Cockburn… (2001) Manifesto for Agile Software Development https://moodle2019-20.ua.es/moodle/pluginfile.php/2213/mod\_resource/content/2/agile-manifesto.pdf page 1-10 (download date: 2021.11.25.)

[14] Agile Practices https://web.archive.org/web/20140209152034/http://guide.agilealliance.org/ (download date: 2021.11.25.)

[15] P Abrahamsson, O Salo, J Ronkainen, J Warsta, (2017) Agile Software Development Methods: Review and Analysis https://arxiv.org/ftp/arxiv/papers/1709/1709.08439.pdf page 20(download date: 2021.11.25.)

[16] A Sharma, M Kumar, S Agarwal (2015) A Complete Survey on Software Architectural Styles and Patterns https://www.sciencedirect.com/science/article/pii/S187705091503183X page 17, 19, 20, 22, 23, (download date: 2021.11.30.)

[17] N Dragoni, I Lanese, ST Larsen, M Mazzara, R Mustafin, L Safina (2017) Microservices: How To Make Your Application Scale https://arxiv.org/pdf/1702.07149.pdf page 1, 2 (download date: 2021.11.30.)

[18] D Shadija, M Rezai, R Hill (2017) Towards an understanding of microservices https://arxiv.org/ftp/arxiv/papers/1709/1709.06912.pdf page 2, 3, 5 (download date: 2021.12.02.)

[19] Y. Niu, F. Liu, Z. Li (2018) Load Balancing across Microservices https://newypei.github.io/files/infocom2018.pdf (download date: 2021.12.07.) page 2.

[20] Agile Alliance: Extreme Programming (XP) https://www.agilealliance.org/glossary/xp/#q=~(infinite~false~filters~(postType~(~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'xp))~searchTerm~'~sort~false~sortDirection~'asc~page~1) (download date: 2022.02.07.)

[21] Agile Alliance: Scrum https://www.agilealliance.org/glossary/scrum/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'scrum))~searchTerm~'~sort~false~sortDirection~'asc~page~1) (download date: 2022.02.07.)

[22] Agile Alliance: TDD https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1) (download date: 2022.02.07.)

[23] M. Fowler (22.04.2020): DomainDrivenDesign https://martinfowler.com/bliki/DomainDrivenDesign.html (download date: 2022.02.07.)

[24] S. Newman (2021): Building Microservices Designing Fine-Grained Systems 2nd Edition O’Reilly Media Inc., Sebastopol, Canada, page 89-197 ISBN 978-1-492-03402-5

[25] M. Steen, A. Tanenbaum (2017): Distributed Systems, 3rd edition CreateSpace Independent Publishing Platform, Scotts Valley, CA, USA, ISBN 978-90-815406-2

Antonio Bucchiarone Microservices Science And Engineering Springer

https://d1wqtxts1xzle7.cloudfront.net/67348420/Antonio\_Bucchiarone\_et\_al\_Microservices\_Science\_And\_Engineering\_Springer-with-cover-page-v2.pdf?Expires=1639956065&Signature=celBWYEn1OPRHNQCsJ6GF4N-WnnEET-puwnl~W8lBZVmkv8WtZ1Vrpn3OzKtGsgZlu1UIrctbEPFyy35tIFAFcSq73aFVWdAR~E4MqHf~wVaUFhAvpLva8l9-BtmLdvrooytB1IF1PYymxkTPEYpxf0B454XD2olk0j~jFesivnvxYYXo6TmLNfDX08uUimnNibZuR5Ek93QEis-gQ0s8nmvpKwxU1qI5SO8Ls10wSUbkVrZtH7ewSLPSnsTg3pO9w1OoiaZhHHIUCpQpV8IKlvlCi7zlfmRs8F44GU8tUe2ru-VU6puDDc2D6OgJ3TjUbf9modZVnW9OUkzROcjuQ\_\_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA#page=136 page 137//129 printed on page