Insight into trading limits in financial algorithms

Bachelor Thesis Computer Science at Optiver

25-06-2018

TU Delft

optiver

Insight into trading limits in financial algorithms

Bachelor Thesis Computer Science at Optiver

By

Eric Cornelissen
Joost Verbraeken
Cornel de Vroomen
Nick Winnubst

in partial fulfilment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Monday July 2, 2018 at 4:00 PM.

Supervisor:	Dr.ir. G. Gousios	TU Delft
Thesis committee:	Dr. ir. Huijuan Wang,	TU Delft
	Ir. O.W. Visser,	TU Delft
	Ing. K. Manios,	Optiver Amsterdam
	Drs. D. Martens	Optiver Amsterdam

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This report describes the project we did for the financial liquidity provider Optiver to conclude our BSc. Computer Science at the TU Delft. Besides describing what we did over the last few months, this report also serves as documentation of our program for Optiver which they can use when they want to know how the different log files are structured and how our program works. The project took roughly 10 weeks to develop and was about solving the problem of verifying risk limits programmatically by creating a software solution that can do this in a reasonable time frame.
Creating the program proved to be quite a challenge because of unforeseen problems and numerous exceptions, but we and the client are quite satisfied with the result.

We would like to thank our supervisors at Optiver for providing us with this project and for helping us along the way. We would also like to express our sincere thanks to Martin, Ian and Alex who were all very patient with answering our questions. Finally, we would like to thank our coach from the TU Delft, Georgios Gousios, for his ideas on how to solve the problem and for providing his guidance with the project in general.

Eric Cornelissen
Nick Winnubst
Joost Verbraeken
Cornel de Vroomen

Delft, June 2018

Contents

Summary

In this project we aimed to create a post-trading day safeguard system that allows for the identification of bugs in the primary and secondary risk control systems at Optiver. These systems are needed to prevent undesirable exposure to the market from happening, and to ensure that they know exactly what this exposure is. The amount of input data for this project, given in the form of log files, equates to roughly 200 GB per trading day, post sanitation. We have developed a program that can simulate an entire trading day and detect if any limits were breached. This program can be run overnight, allowing for a T+1 report in the morning after the respective trading day. The difficulties in this project were in the acquisition of all knowledge concerning the unique traits of various markets around the world, inconsistencies in the data, incomplete documentation, and optimization of the program to run within the required time.

Organizationally, the project was executed within an agile workflow, with Kanban as software development methodology. Furthermore, the project is tested extensively to ensure the accuracy and correctness of the program.

Concerning the impact of the project, it contributed to the identification and resolution of multiple previously unknown bugs in the control systems at Optiver. Furthermore, our project verified the existence of some previously known issues. In the future, when the software is run to verify all order logs of Optiver, the software will prove its value by either increasing the confidence that there is an absence of bugs in the RiskGuard and autotrading software of Optiver or by identifying breached limits, indicating a bug.

1. Introduction

Optiver is a liquidity provider on the global financial markets. They trade the markets with their own money and are always willing to sell or buy financial instruments to or from you. Trading all these hundreds of thousands of products is impossible to do by hand. Instead, Optiver uses sophisticated algorithms which automatically keep track of the vast number of trades and products. To make sure that these algorithms do not get out of control, certain limits are imposed on them. There are systems in place which monitor these limits, and they should guarantee that orders that would breach these limits are not sent to the exchange. To make sure these limits are actually adhered to, we created a program to analyse historical data and check if any breaches have occurred. Initially on data from the 7th of February until the 16th of April, but it should also usable as a daily check. The biggest challenge was to create a good simulation of the orders and to include all the different edge cases. Also, doing this in a reasonable timeframe was something we had to constantly keep in mind while creating the program. The initial time frame, which consists of about 60 days, has in total about 12 terabytes of data, which is about 200 GB of data per day.

Besides creating the verification program, part of this project was also to evaluate Optiver's datasets, which are mostly generated for compliance reasons, and to report any inconsistencies. While parsing the files we did find missing data and inconsistencies, which are useful for Optiver to improve their logs.
Our coach is Dr Georgios Gousios from the TU Delft who specializes in Software Analytics. This field helps software developers to get quantitative statistics about their programs and use these insights to create better quality software. We've asked Georgios to be our coach because of his extensive knowledge about software engineering and data science, two relevant fields for this project.

Optiver, represented by Kris Manios and David Martens, is a leading financial liquidity provider and market maker focused on pricing-, execution- and risk management. Optiver provides liquidity to financial markets using its own capital, at its own risk, trading a wide range of products: from listed derivatives to cash equities, and from Exchange Traded Funds (ETFs), and bonds to foreign exchange (Fx). Optiver's independence allows them to objectively improve the markets and provide efficiencies for market participants.

2. Problem Definition

Our economy is dependent upon the value added by companies. These companies rely on investments in order to have the money to grow. These include simple loans, bonds and the selling of shares of the company. For investors, it is important to always be able to sell and buy bonds and stocks in the companies they want to invest in, for this reason such financial instruments are generally traded on stock markets. It is also important for them that the difference in price for which they can buy and sell these instruments (i.e. the "spread") is as small as possible and that there is always enough of these instruments in the order book, both on the bid and the ask side. This is exactly the goal of Optiver as a market maker.

On the stock market, the spread is low and the number of instruments being offered is quite large. However, on the option market, this is normally not the case. For every underlying product, there is a huge amount of options available at different strike prices and expiration dates. Unfortunately, the spread is generally quite high and the market quite inefficient because there are few traders for any given option. Optiver aims to solve this problem by acting as a market maker that is specialized in options and sells and buys instruments for prices that are reasonably close to each other. Optiver needs to have a good estimate of the theoretical price of an instrument and competes with other market makers to buy and sell effectively.

Constantly putting in orders and monitoring them is done using algorithms, which can do this fully automatically. These algorithms could potentially make mistakes, for example when something unforeseen happens or when there is a bug in the code. Risk monitoring systems are needed to constantly check the risks that these algorithms take. Orders which would breach these limits are not sent to the exchange, hopefully preventing excessive losses.

Optiver wants to know if these limits are implemented correctly by verifying the order logs, and checking if any limits were breached without the risk systems noticing.

- If we don't find any breaches, Optiver has a stronger guarantee that their risk monitoring systems work correctly.
- If we do find breaches, then we can derive from the breaches where the bugs that are causing these breaches are present in the software of Optiver. Identifying these bugs adds tremendous value to Optiver because it is very risky to trade when bugs in their systems are present.

Primary problems

The primary problem can be seen as two smaller sub-problems, namely we need to give Optiver the capabilities to verify the order logs against the limits for:

- A single day of trading
  - Needed for a daily run over the data and doing the limit verification in a T+1 manner
- A range of dates
  - Needed for an historical evaluation of the log files

Secondary problems

We need to give Optiver:

- The option to analyze a specific market instance, instrument, underlying, user, limit role, or type (order or quote).
  - Needed when Optiver wants to investigate breaches only for certain entities.
- An implementation of the program that can analyze a single day within (ideally) 8 hours.
  - Needed because Optiver wants to have in the morning a report on the breaches of yesterday, and the time frame between the last log entry and when the first developers come in is approximately 8 hours.

In the section 5. Value Proposition, we give an extensive overview of how we will add value to Optiver, which is done primarily by solving these problems.

3. Problem Analysis

3.1. Overview

For this project, we analyzed day-to-day log data generated by software systems at Optiver to see if any trading limits set by either Optiver or the exchange, are breached. This includes a) the logs of the systems that communicate with exchanges, containing details about all the trading done by Optiver, called the order logs, b) an overview of the limits and how they change over time, called the limit history, c) the logs of an internal system called RiskGuard (which already verifies the trading limits in real-time), called the RiskGuard logs, and d) a set of logs that contain specific internal definitions, called the audit records. By combining the data found in these four files (for details see the table in the 3.3. Overlap in input data section) it is our task to determine if at any point in time any trading system at Optiver ever exceeded a limit without Optiver knowing about it.
The software system built during this project will have two main purposes. First of all, the system should be able to do a historical analysis of the order logs secondly, it will be used on a day-to-day basis to analyze the previous day of trading to be sure that their systems are valid. The first use-case requires us to make the program easily scalable (be it multi-processing or distributed computing) so that all the terabytes of data accumulated in recent years can be analyzed in a reasonable time frame. The second use case requires us to make an efficient and quick program in general, as this would allow Optiver to run it on a simple PC, rather than a dedicated machine, in a daily fashion.
As an output Optiver expects a list of breaches, if any, that occurred. Besides that, we will also output a list of any warnings or errors that occurred during the analysis which will expose problems in either our program or in any of Optiver's logging systems.

3.2. Input Files

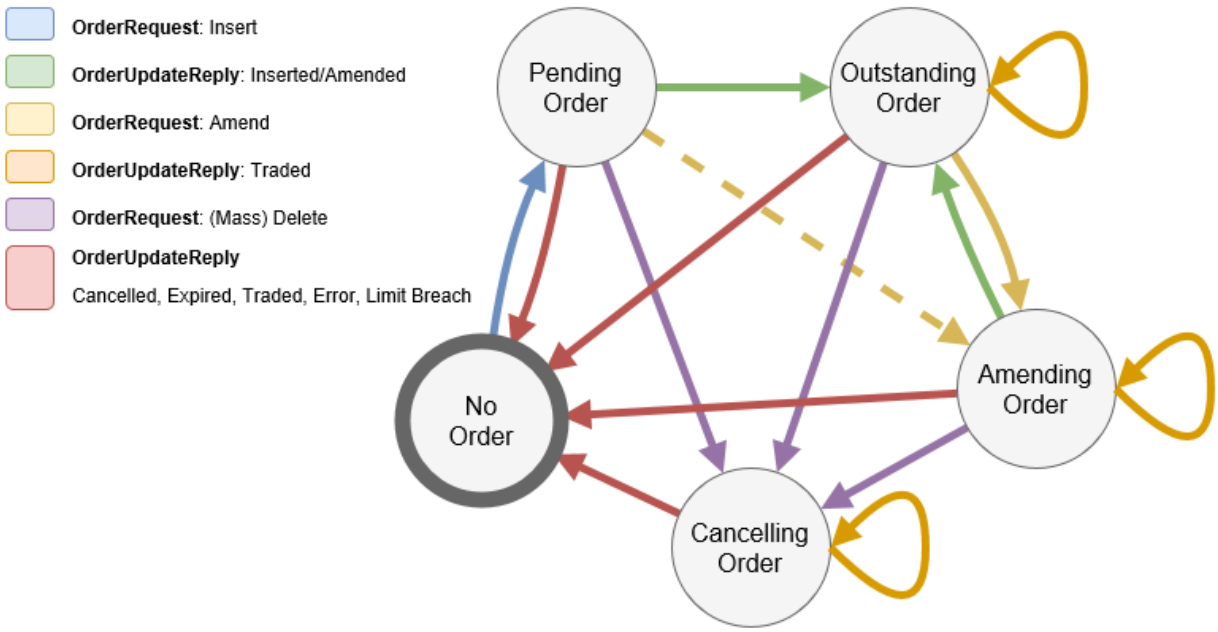
In this section, the 4 types of input files are analyzed and described. The datasets we are using, as mentioned before, are the order logs, the limit history, the RiskGuard logs, and the audit records.

3.2.1. Order logs

Description

The order logs consist of all the orders sent to exchanges and their respective responses. Every time an order is sent to an exchange, the attributes of that order are logged. When the response of the exchange is received that indicates, for example, that an order is placed in the order book, another logline is created. Finally, messages from the exchange that indicates for example that an order is expired or cancelled are also written down in this log.
A snippet of the loglines in from the order log can be found in Appendix B.1.

The log is in a key-value pipe delimited format. Every line either involves a request to an exchange or a reply from an exchange. Because the type of the message can differ per line, they do not necessarily have the same attributes. The files contain either orders or quotes, where an order is a single-sided order that is mostly used to directly try to buy or sell something, and a quote is a type of order which stays in the order book often having two sides, a bid and an ask.
A simulation of these orders over time must be made such that we know what order conditions (e.g. outstanding volume or amount of active orders) there were at any moment. For calculating the value an order has on a certain limit we need the attributes like VOLUME and PRICE. The attributes MARKET and FEEDCODE are needed to map an order to the limits and to the product specification. Finally, the timestamp at the start of the line is critical to check which limit was active at that moment and to verify time-based limits. The feedcode is an internal code used to identify products. The underlying product is not directly present in loglines, and since some limits are given per underlying, we need a mapping from feedcode to underlying, which is what the audit records will be used for.
An order can be in multiple states, such as pending, amending and traded. It can also change from some state A to some state B. An interesting challenge for us is that the transition is not instant, so the state transitions themselves are also states in our simulation. An example of this is amending an order: when Optiver sends a request to amend an order it takes time before the exchange receives that request and amends it, hence, in the meantime the state of the order is amending rather than amended. A state machine representing the different states of an order is illustrated in the following diagram:



Files

The order log files are gzipped and structured in directories based on the year, month and day. The size of these compressed files ranges from several kilobytes to 300 megabytes, with a compression ratio of 7%. This results in files of up to 4 GB when these files are uncompressed. If a log file reaches 4 GB for one day, a new log will be started in a new file continuing from that point onwards. For the analysis we have to keep in mind that we should regard all these separate log files as a single big log file.
The filename of a log (e.g. "20180302-1830-20180302-0830-afex_co_at_aucm_201.OrderLoggingAudit") contains the start and end time of the log, like in the example where the log is from 08:30 to 18:30. It also contains the code of the operating market link and its instantiation number. Note that market links can have multiple instantiations. For our project, all limits are separate per market link instantiation.
The logs can contain 7 types of log entries, namely: *OrderLoggingInfo, OrderRequest, OrderUpdateReply*, *QuoteRequest*, *QuoteUpdateReply*, *MassDelete*, and *Position*.

- The type "*OrderLoggingInfo*" indicates the start of a logging file and contains some general information.
- Orders with the type "*OrderRequest*" are sent by Optiver and contain details about the order placed, such as the market, volume, price, and lifespan.
- Orders with the type "*OrderUpdateReply*" come from the market and inform about what happened to a particular order (identified by the request id), for example if the order is inserted, expired, or traded.
- Orders with the type "*QuoteRequest*", similar to order logs, are sent by Optiver with information about the quote.
- Orders with the type "*QuoteUpdateReply*", similar to order updates, come from the market with updates about a particular order (again identified by the request id).
- "*MassDelete*" entries are requests to quickly delete a group of quotes.
- "*Position*" log entries contain the position in an instrument for a particular moment.

Irrelevant data

Some loglines of the order logs don't need any processing and can be ignored safely. These types of loglines are:

- Replies from the exchange to confirm that an order is inserted. Because the limits assume that an order is already in the order book as soon as it is sent to the exchange.
- The request to delete an order. Because there are no limits on the number of delete operations / the order can still be traded before the reply from the exchange is received.

- The request to mass delete all orders. Again, because there are no limits on the number of delete operations.
- The current position of Optiver in their orders, indicated with "Position". Because we're already keeping track of their current position based on the order requests/replies.
- The starting line of the log file, indicated as "OrderLoggingInfo". Because we only care about the actual logging statements about the orders.

An overview of all the attributes for the loglines can be found in Appendix F. Order data overview.

3.2.2. Limits

Description

To check if any breaches have occurred, we need to know what limits were active at any point in time. To get the limit we need the market, underlying, trading type (column limit_role), limit role id and the timestamp. With the market, underlying, trading type and the limit role id we find a list of limits for this combination. With the timestamp, we compare the start and end date times of these limits to find the active limit at the given time. The limits file is CSV formatted and a sample can be found at Appendix B.2.

Most of the 35 columns are relevant for the purpose of our project, but some can be ignored, like update_user and update_comment. The entire log file contains about 300,000 rows, so it can easily be read sequentially and kept in memory during runtime, as it will occupy just ~78MB of RAM.

Files

All this data can be found in a single file that is CSV formatted. The so-called *limits history* file contains a list of all limits sorted on time. We identify a certain limit using the combination of the market, underlying product, trading type, active role and the time.

Overview of the limits

The following limits are checked by Optiver:

- Limits checked per request log line
  - *Maximum quote volume*
  - *Maximum order volume*
  - *Maximum order value*
- Limits checked over all outstanding orders
  - *Maximum number of outstanding orders*
  - *Maximum gross volume outstanding orders*
  - *Maximum gross value outstanding orders*
  - *Maximum outstanding volume per instrument*
- Limits checked over time intervals
  - per underlying
    - *Maximum number of quote insert/amend/delete operations*
    - *Maximum number of insert/amend operations*
    - *Maximum gross traded volume*
    - *Maximum gross traded volume over a long term*
    - *Maximum gross traded value*
  - per instrument
    - *Maximum number of quote insert/amend/delete operations*
    - *Maximum number of insert/amend operations*
    - *Maximum gross traded volume*

All of the limits that are checked over time intervals have a window size that specifies the time interval in seconds.

3.2.3. RiskGuard logs

Description

RiskGuard is an internal system at Optiver that (among other things) accumulates the limits from the respective database on startup and can then be called by other systems to request access to a certain limit. Before an auto-trader starts trading, the market instance to which it will be sending orders will retrieve the corresponding limit data based on what the auto-trader wants to trade and the role it wants to have. The RiskGuard has a log file for all of the operations it does and contains information like in the snippet below (data is randomized). Besides the limit roles, the foreign exchange rate and internal markets (markets that we do not have to analyze) are also retrieved from the RiskGuard logs. The RiskGuard system creates a single log per day, and there are always two RiskGuard systems active at any time. The two RiskGuards communicate with mutually exclusive subsystems within Optiver, so there are two separate log files per day. A snippet of the RiskGuard logs can be found in Appendix B.3.

Files

The RiskGuard logs comprise 2 gzipped plain text log files per day, one log for each RiskGuard system.

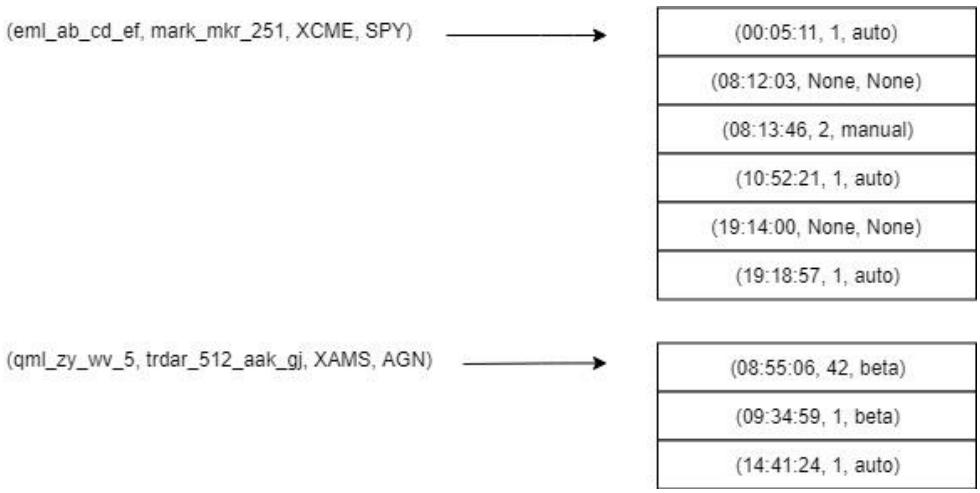
Data

Limit roles

To verify any trades found in the order logs the timestamp from that trade is used to find the most recent limit assignment for the market instance, user, market, and underlying combination belonging to the order. This results in a role id (role_id, as seen in the snippet above) which is then used to retrieve the corresponding limit values from the limit history.

To do this the parser for the RiskGuard log files focuses on "*Receive LimitUsageStatus*" log entries. Such an entry is expected to have a *market*, *underlying*, *trading type*, *role id*, *username*, *market instance name* and *status* specified. The *market*, *underlying*, *username* and *market instance name* form together the primary key for a list of tuples of roles and trading types. For each entry with the given key, a tuple containing the new role id and trading type is added to the list together with the timestamp of the log entry. The status found in the entry specifies whether the role was added ("*USAGE_ACTIVE*") or removed ("*USAGE_INACTIVE*"). If it was removed, a tuple is added to the list containing only the given timestamp. Then, if a role id or trading type is requested, the list will be searched for the corresponding timestamp. If there was no role active at the requested time an error is thrown because this should never happen and because we are unable to find a limit without a role.

Figure 1



The figure above is showing an example with two lists of roles. Both lists have a key as a pointer containing the instance name, username,

market and underlying. The list itself is ordered on timestamp and contains tuples with the timestamp, role id and trading type.

Foreign Exchange rates

The orders placed by Optiver are sent all over the world and are paid with many foreign currencies. The limits values are all set in euros, so these orders in a foreign currency should be converted to a value in euros. The foreign exchange rate (or *fx-rate* for short) is needed to calculate the value for an order. These rates not always accurate since the actual rate changes every moment of the day while the RiskGuard logs might only have a single fx-rate for a currency per day, but this is the most reliable data for the purposes of this project.

Internal markets

For our analysis, we should ignore all internal markets because most limits do not apply there. To find out what the internal markets are at any given day, the parser for the RiskGuard log files looks for "*Receive InternalMarket*" log entries. These log entries are structured similar to the "Receive LimitUsageStatus" seen above, but the expected fields are the market instance code (mic) and the action for that line in this case. If the action specifies that there is a new market instance that should be added ("*ACTION_ADD*"), then the market instance code, which is an identifier for the market, is added to a list of internal markets. These internal markets are added once at the start of the day, and the list does not change during the day.

3.2.4. Audit records

Description

The audit records are log files in JSON-format that are generated by an internal monitoring system. Every time Optiver starts trading a new instrument, their systems add an entry to the audit record that links the instrument to the corresponding attributes of the instrument. This includes for example the market on which the instrument is traded, the feedcode, the underlying product and the type (*spot*, *option*, *Fx*) of a product. We are using the audit records because they contain data that is necessary to retrieve the relevant limits, which are identified by i.a. the underlying product and kind of order. One log entry of the audit records looks like the following (simplified). In Appendix B.4 a larger example of the audit records is shown.

```
{
	"message": {
		"instruments": [{
			"instrument_id": {
				"feedcode": "XETR.NL0000370179.EUR",
				"market": "ZOLD"
			},
			"kind": "FUTURE",
			"future_attributes": {
				"underlying": "MOO",
				"multiplier": 1,
				"currency": "EUR"
			}
		}],
		"market": "ZOLD"
	},
	"message_type": "MSG_INSTRUMENTS_BY_MARKET"
}
```

Files

The audit records are very simple to parse. Every day has exactly one corresponding audit records file which can be parsed trivially after unzipping, because every line in the file is in JSON format.

Overview of the fields

- **message**: object

    - **instruments**: list of objects
        - **instrument_id**: object
            - feedcode: identifier of the instrument, corresponds to the feedcode found in the order logs
            - market: the market on which the instrument is traded
        - **kind**: enum, must be one of: Bond, BondFuture, Call, Combo, DividendFuture, ETF, Future, Fx, FxForward, LEPOCall, Put, Spot, VolatilityCombo
        - **<kind>_attributes**: class, e.g. future_attributes or spot_attributes
            - **underlying**: underlying product
            - **multiplier**: multiplied by the temporary value to get the final value of a product
            - **currency**: 3-character string identifying the currency being traded
    - **market**: the market on which the product is traded
- **message_type**: Indicates the type of the message, for us only "MSG_INSTRUMENTS_BY_MARKET" and "MSG_INSTRUMENT_INSERTED" are relevant.

3.3. Overlap in input data

The different data files can be linked to each other through shared identifiers. The table below shows relevant data points for us that we can use to combine data in order to analyze it. “Key” indicates that we need the attribute to find the value. “Value” indicates that the attribute can be found in the corresponding file. For example, the timestamp is found in the order log and is needed to get data from the Risk Guard / Limit history.

	Explanation	Order log	Risk Guard	Limit history	AuditRecords
Timestamp	Limits change over time, so this is required to determine the limit	Value	Key	Key	
Market	Orders are to be analyzed per market-underlying-user combination	Value	Key	Key	Key
Feedcode	A code representing the market and underlying	Value			Key
Trading type	The type of trader for which this limit applies, i.e. "manual", "auto", or "beta"		Value	Key	
Role id	The current role of the trader		Value	Key	
Username	Orders are to be analyzed per market-underlying-user combination	Value (only for requests)	Key		
Instance name	The instance name of the algorithm	Value (in filename)	Key		
Underlying	Orders are to be analyzed per market-underlying-user combination		Key	Key	Value

3.4. Output files

For each logline in the order log, the knowledge base is updated, and the limits are checked. Every time a breach is found, or an error is thrown for a certain order logline, this is logged. The output files of the program consist of the log with breaches, the log with errors, and the log with warnings.

Location of output files

When the program is started, a root directory with the data must be defined. In this root directory, a new directory 'output' is created. For the date that is run, a directory is created which will contain 3 inner directories, namely 'results', 'warnings' and 'errors'.

Breaches-log

When the program has found a breach, the most important information about the order is logged so that Optiver can research the breach. For each order log of a market link, if any breaches are found, a CSV file is created with the following columns and one row per breach. Each breach is written as soon as it is detected so that they are not lost in case the program crashes.

DATE	*TIME STAMP*	*MARKET*	*UNDERLYING*	*INSTRUMENT*	*ROLE _ID*	*TRADING_TYPE*	*USER*	*BREACHED _LIMIT*	*LIMIT _VALUE*	*REAL _VALUE*

The *date* and the *timestamp* are added to know when the breach happened. The *market*, *underlying* and the *instrument* are used to identify which product and market the breach occurred on. The *role id*, *trading type* and *user* are needed to find out which limit should have been active at the moment of the breach. Finally, the *name* of the limit and the *maximum value* of the currently active limit and the *real value* that exceeded the limit are logged.
The name of the breaches-log starts with limit_breaches and contains the date and the market instance name.

Warnings-log

The warnings-log contains all the exceptions thrown in our program that are thrown because of a mistake in the data itself, instead of a bug in our program. This means that for example replies from exchanges with no corresponding request found are logged here. Also when orders remain outstanding at the end of the file, they are logged as warnings.
Warning loglines are derived from exceptions. The type and the message of the exception are logged, together with the timestamp of the order log where the exception is thrown.
An example warning:

```
11:31:39.176798 - [<class 'AssertionError'>] ERROR WHILE EVALUATING FILE: No
outstanding order matching the reply: order={'__LOG_DATE__': datetime.time(11, 31, 39,
176798), 'TYPE': 3, 'TRACKERID': 1476, 'VOLUME': 150.0, 'PRICE': 3993.5, 'TIMESTAMP':
1519381899176773000, 'MARKET': 'CHIX', 'FEEDCODE': 'XLON.GB0007188757.GBp',
'CHANGEREASON': 6, 'KIND': 'SPOT', 'PRICECCY': 'GBp'}
```

Errors-log

The errors-log contains all the exceptions thrown which are not caused by a fault in the data that we know of. For example, these can be errors made by us in the program or unexpected errors in the data which are not already handled in our program. Similarly, to logging the warnings, the type and the message of the exception together with the timestamp is logged.

4. Requirements

After analyzing the problem, we drew up the requirements for the project. The requirements are prioritized using the MoSCoW-method (Tudor, 2006), which separates must-haves, should-haves, could-haves and won't-haves.

Must Haves

- Correctly verify the limits
  - It is important that the limits are verified correctly and do not give any false positives or false negatives. Correctly implementing this requires us to achieve a very high accuracy and correctness which is done by carefully creating the code and reviewing that of others thoroughly. Besides that, creating tests to infer confidence in the correctness of the implementation is key to achieve this requirement.
- Generating an overview of all breaches with the relevant data
  - Only knowing there has been a breach does not provide a lot of value; to research the breach we also need for example the timestamp, the limit that was breached and the observed value.

Should Haves

- Being able to verify one day in a ‘reasonable’ timeframe
  - A daily verifier is the target deliverable of the project. To let our program evaluate every day continuously, it should be able to evaluate an entire day of data in less than 24 hours. However, the client prefers that the program has finished the evaluation of the day before when the developers come into the office in the morning. Therefore, the goal is to run the program in about 8 hours.
- The ability to notice errors in the data
  - There could possibly be errors in the large datasets which should be logged to a file so that they can be researched. These errors could indicate bugs in the data generation code or have something to do with errors from the exchange.

Could Haves

- The ability to run the verification of the logs in (near-)real-time
  - This way breaches can be noticed and resolved quickly. Implementing this feature requires creating a verifier running daily on the servers of Optiver.
- A GUI showing the breaches in a clear way
  - We could make a GUI where the breaches are shown in a visual way. This would also contain statistics about the days run.
- Run on a framework
  - To run the program on a computing cluster and to have features like fault recovery, we can build our program on a framework which does the node orchestration for us.

Won’t Haves

- Won’t run on simulated and internal markets
  - Simulated markets don’t have limits applied, or have specific limits for testing purposes. On internal markets other risk limits apply which are not automatically checked by the RiskGuard, so we don’t have to check these either.

5. Value Proposition

To solve the problem of the client and add as much additional value as possible within the scope of our project, we focused on the 3 things that are listed below. These are ordered based on their priority and extended with several sub-levels of detail.

5.1. Check if limits where breached on a past date

The main purpose of the project was to make a program that gives Optiver some guarantees about the workings of the auto traders and risk management systems. To achieve this, we executed the following steps:

- Parse the limits
  - Parse the CSV file containing the limits
  - Group the limits based on their market, underlying, trading type and role id
  - Sort the limits in the same group based on their starting date
- Parse the audit records
  - Parse the audit records
  - Get from every audit record line the instruments
  - Get the underlying, multiplier, kind and currency of every instrument of every audit record based on the feedcode and market
- Parse the RiskGuard log files
  - Parse the log files
    - If the logline mentions the addition of a new internal market, then store this market in a list
    - If the logline mentions which role is used, then map this role to the correct identifier
    - If the logline mentions an update of the Fx rate of a currency, then map this rate to the currency
  - Sort the breaches, roles and Fx rates based on their timestamp
- Initialize verification classes
  - Initialize the Ledger. The ledger keeps track of all outstanding orders to verify for example "max outstanding volume"
  - Initialize the Ticker Tapes. The ticker tapes keep track of how much the volume/value/number of operations changed in a certain period of time for a certain instrument
- Analyze all loglines, for every logline:
  - Skip if the logline is irrelevant
  - Get the relevant feedcode data
  - If the order is a reply to a request, get relevant data from the corresponding request
  - Get the relevant RiskGuard data (trading type, role id, multiplier, Fx rate)
  - Get the relevant limits
  - Update the window size of the relevant ticker tape(s)
  - Calculate the value of the order based on i.a. the price / volume
  - Update the ledger
  - Update the ticker tape(s)
  - Verify if any limits imposed on the order are breached (e.g. "max volume")
  - Verify if any limits imposed on the ledger are breached (e.g. "max outstanding volume")
  - Verify if any limits imposed on the ticker tape(s) are breached (e.g. "max volume traded per second")
  - Log all breaches/errors
- Check if the ledger is empty
  - If not, throw an error

This is a solid approach to the problem, as it checks the order logs against the limits. But more importantly, we have to show that this plan indeed works by creating tests that verify that no false positives will be

reported, and no false negatives will be forgotten. In other words, that there are neither false positives nor false negatives in the detected breaches.

5.2. Check the limits in a reasonable amount of time

We have had to guarantee that the limit verification can be done in a reasonable amount of time. Ideally, verifying one day would take a couple of hours, as the verification can be started at midnight, and the output can be observed in the morning when the developers come in. Therefore, we have had to optimize the execution speed of the program. The log files add up to about 200GB per day. That amount of data takes a non-trivial amount of time to analyze, especially if one is not paying attention to the execution speed. Therefore, we considered parallelizing the process, which will be elaborated in the corresponding chapter.

5.3. Inspect the log files and report all inconsistencies

Our project is the first one that extensively uses the log files at Optiver for analysis. The normal purpose is just to being able to show the financial regulator what has happened at any given time. Because the files are not regularly used for analysis, it could have inconsistencies. Discovering these inconsistencies is valuable for the developers, as there are mandatory rules for how these log files should look. Hence, our program will also test the log files and internal systems of Optiver for inconsistent, incorrect or unexpected output. As a result, we will add value in a trivial way because the parsers we write must parse the log files anyway, and we need the consistency and correctness in order for our program to work correctly.

6. Workflow

In this section, we will delve into how we approached the project on a day to day basis. It gives insight into how we worked, what software development methodologies we used and how we divided the workload. It also explains how we communicated, both within the team and with our coaches.

6.1. Software development methodology

For this project, we decided to use the Kanban methodology (Anderson, 2010) for our workflow. This choice was made because we wanted a workflow that allowed us to work flexibly, balance the workload of the different parts of the project, and switch priorities as needed. Kanban, as opposed to more formal systems such as Scrum, allowed us to do this by not requiring us to make a planning every single week, but instead whenever the requirements changed or when we finished a phase of our project. When a team member is done with a task, he simply picks up the next most important task to work on.
The primary reason we needed this flexibility was that of the nature of the input data for this project, which varies in size and format significantly. This regularly led us to discover minor but critical issues by running our program on different subsets of the input data. Kanban allowed us to switch priorities effortlessly, and make sure such issues are resolved as quickly as possible. Furthermore, we often had to wait on Optiver employees to manually verify a breach or error when we felt confident that we discovered one in a system used at Optiver. By using Kanban, we can simply report the issue and continue working on the next most important issue.

6.2. Git and Pull Requests

As a version control system, we used Git together with BitBucket. This is the repository hosting service used by Optiver and stores the code on an internal server. We used 2 main branches: master and develop. The master branch contains the releases of the program, which was used to share with the team at Optiver. The develop branch contains the version that is still in development but should be working at all times.

When adding a feature, a new branch was created from the develop branch and after implementing the feature, a pull request (or PR) was opened to merge that branch into the develop branch.
For each opened pull request, a thorough review was executed to improve the software quality and the correctness of the program. When at least 2 people approve a pull request, the pull request can be merged.

Continuous integration: we used continuous integration to guarantee that our program works correctly. This was done by requiring a successful build before a PR can be merged to the main branch. We used Bamboo for the continuous integration, which is a very flexible framework that we configured to run all tests in an isolated execution environment and to report any compilation failures.

6.3. Daily stand-up

We worked together quite a lot every day and we think that a daily standup is a great way to discuss things with the whole group. Because we worked together closely on the program, and a lot of parts of the program work together, we already knew what everyone was working on. But to have a moment for group discussions and to properly divide the work, a daily standup meeting is very useful. Hence, we did one every morning at 9:30.

6.4. Weekly meeting with client

Every Monday at 1 o'clock we had a meeting with our clients and discussed our progression and asked for / got feedback on our work. On top of that, because our clients were sitting just a couple of desks away, it was really easy to just walk by and ask a question.

6.5. Code styling

We focused a lot on writing “clean code” and used both automated static analysis and manual analyses to improve upon this.

Automated static analysis:

- **Style:** to guarantee a consistent style in all our code, we used the standard style guide from the PyCharm IDE. The PyCharm IDE comes with a clearly defined set of rules which we could all follow to guarantee the code base is readable and consistent. Since the PyCharm IDE allows you to customize the rules that you must adhere to, we occasionally made changes to the preferences in order to make it comply with our preferences as a team. These changes are always discussed with all members of the team, and are then also applied by all members of the team. For example, we increased the maximum line length from 80 to 120 tokens because this is more readable, and our screens were wide enough for this change.
- **Potential bug elimination:** we used several static analysis tools to prevent bugs from happening:
    - **The built-in static analysis tool from PyCharm** shows warnings and errors on places where the code can be improved or is incorrect. For example, unused variables, unreachable code and redundant parentheses are already marked and solved by the IDE.
    - **PyLint** is static analysis tool that adds a few extra warnings, including redundant else statements or implicit return statements. We did modify several rules including the line distance (from 100 to 120 tokens) and the empty line at the end of files.
    - **Python 3 typing hints** allowed us to partially overcome the non-type-safeness of Python. This enabled the IDE to show warnings where the type of the arguments does not match the signature type of the parameters of the methods. All our methods have their parameter types and return type declared by using these typing hints to reduce the chance of bugs due to passing incorrect types as much as possible.

Manual static analysis: we made extensive use of pull requests to review our code and kept them short so that they are easy to review. The project was developed with a total of approximately 400 pull requests, all of which have been checked by the continuous integration system, reviewed and approved by at least 2

fellow group members before they were merged in the develop branch. Our supervisors also reviewed our code by means of a pull request in the second-to-last week and provided us with some useful tips.

6.6. Division of labour

Dividing the work among the team members mostly came naturally. Because of the nature of Kanban no-one was ever out of work and everyone had something to do during the entire project. Every member of the team contributed approximately the same amount of time as the others on writing code, writing tests, writing documentation, writing reports, researching, and peer reviewing (in the form of code reviews for pull requests). Occasionally, we reflected on the past few days of work and decided to switch tasks to give everyone a chance to work on every part of the project.
As for big and/or complicated tasks, the person working on them was often chosen after a brief discussion with all team members to make sure that everyone knew to whom they should ask related questions. Often, such tasks even involved 2 developers doing pair programming or splitting the problem up into smaller pieces and dividing the work.
The final product is divided into several major (and mostly disjoint) components, all of which were either made or reviewed extensively by at least 3 of the 4 team members. This way we ensured that we all understood the entire code base which was important to learn as much as possible during this project.

6.7. Communication within the team

Within our team, we communicated primarily in person because we had our own joint workspace at Optiver. Hence, we could talk to each other directly and there was no strong incentive to communicate over email or other electronic applications. Due to the impossibility of working on the project on an external computer, combined with considerations of our NDA, there was neither need nor incentive to communicate details over external digital media. We did use WhatsApp to communicate delays or unexpected absences. If a team member was off for a meeting or something else, he would get a quick recap of what had happened while he was away from the other team members after returning to Optiver.
We held a stand-up meeting every day of the workweek with all team members that were present to discuss what happened the previous day, what we planned to work on that day, and if there were any problems. These meetings helped us all stay on top of what was going on within the project, enabling all of us to explain our current state and course of action to anyone interested. It also ensured that everyone knew who was the right group member to ask his question to, that we did not do the same work twice, and work could be redistributed if we found that the amount of work was no longer properly balanced.

6.8. External communication

Outside of the team, we had two primary communication partners, namely our coaches at the TU Delft and Optiver, and a number of other individuals within Optiver with whom we communicated about their internal software systems. With our coaches, we communicated primarily by using emails, Skype and in meetings. Our coach from the TU Delft we met on a semi-regular basis, approximately once every 3 weeks. With our coaches at Optiver we met at least once a week for a quick catch-up, occasionally we would meet up a second time for more in-depth discussions. Besides that, we could always approach each other with questions throughout the day, either by mail or in person.
Besides our coaches, we also interacted with many different people within Optiver. The primary reason for this was because of issues found by us in their systems. In such cases, we reported to our coaches at Optiver who referred us to the person or team responsible for the corresponding piece of software. This communication was usually initiated through a simple email and based on the severity and scope of the issue, which could be followed up with a meeting which is then attended by the team member who discovered the bug.

7. Design

In this section, we will look at the architectural design of the final product and explain why the design is justified for our problem. We will also discuss how our design decisions changed during the project.

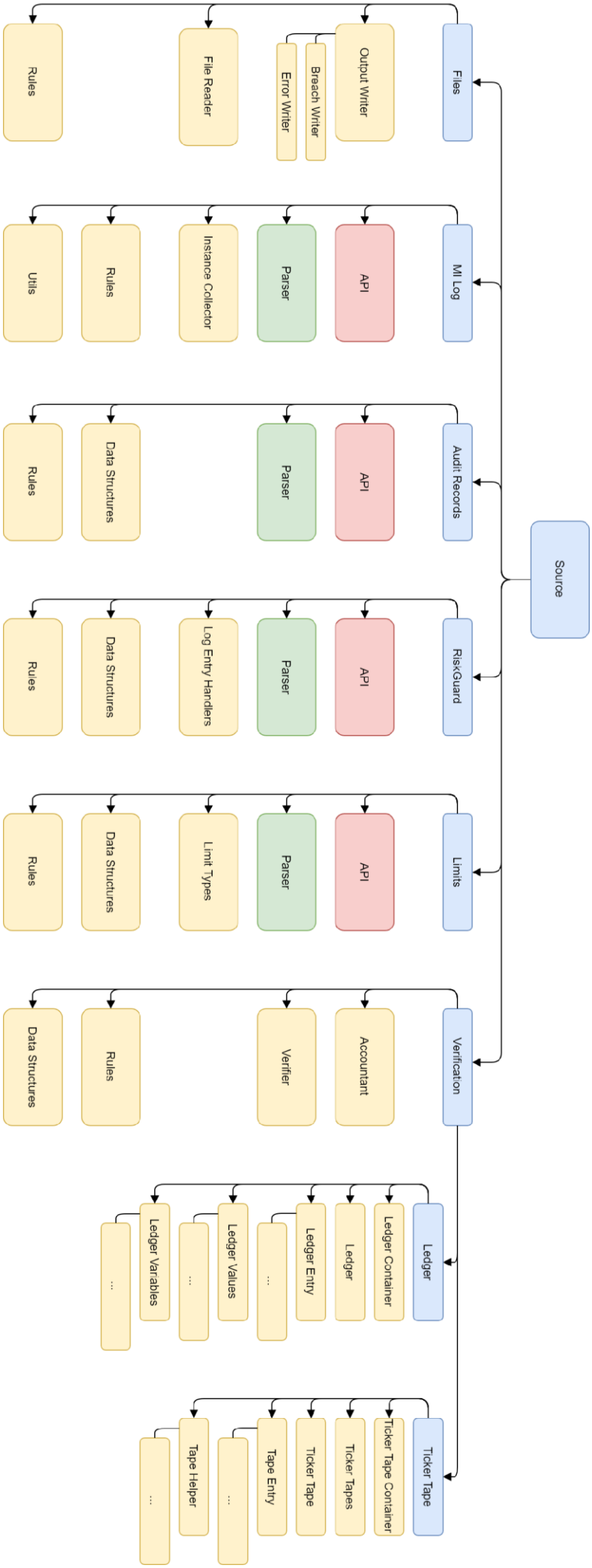
7.1. Architecture

The program is split into a total of 6 different parts. One module controlling the execution, one module verifying the data and 4 modules representing APIs for the different input files. These modules are each split up into submodules to keep responsibilities separated.
As the figure 2 below displays, each API module is split into 6 distinct modules.

A short explanation of the most important files:

- Input related files
  - An **API** file exposes the public API which allows other modules to interact with the data it has. It often contains a cache that contains its data to speed-up the API calls significantly at the cost of using more memory.
  - A **Parser** file reads the input data, converts strings to integers, floats, enums, etc. and returns a data object containing the parsed data. This data object is exposed by the API to the other modules.
- Reusable code (which is separated from the non-reusable code as explained under "Software Quality & Testing")
  - A **Data Structures** file contains the definitions of classes and/or named tuples that represent a plain data object.
  - A **Rules** file contains the immutable data (think of constants, definitions, rules):
    - To improve the safety of the code, the file contains enums because then a typing mistake will result in an IDE warning as opposed to using raw strings
    - To abstract the configuration of the log files away from our actual code, as it contains the mapping of log entries or keys in the loglines.
  - A **Utils** (short for utilities) file contains functionality shared over multiple classes. When only a single class uses a certain method, the method will be placed in that class by the principle of maximum encapsulation.
  - The scripts in the **Files** module abstract the file-system related functionality away from the program. Simplifying for example the logging of breaches and the reading of zipped files
- Main program, i.e. verification, related files
  - The **Accountant** manages the entire process and does not contain a lot of logic. It just calls in the functions exposed by the other modules in the correct order and can be regarded as an implementation of the Façade design pattern.
  - The **Ledger Container** contains all the ledgers that act as the order books which keep track of all outstanding orders. This module is explained in more detail in the section 8.1 Ledger.
  - The **Ticker Tape Container** contains all ticker tapes that keep track of all activity within a certain time frame. This module is explained in more detail in the section 8.2 Ticker Tape.
  - The **Verifier** solves our problem by comparing the values returned by the ledgers and the ticker tapes to the current applicable limits. Essentially all other modules only exist to give the Verifier all relevant data.
  - The **Ledger Entry, Ledger Values, Ledger Variables**, **Tape Entry** and **Tape Helper** classes are created and used by classes in the verification module. Generally, these helper classes are contained in a Data Structures file, but because these helper classes are relatively big and complex they're defined in their own separate files to keep the codebase clean and understandable.

Figure 2


Source
Files
Output Writer
Breach Writer
Error Writer
File Reader
Rules
MI Log
API
Parser
Instance Collector
Rules
Utils
Audit Records
API
Parser
Data Structures
Rules
RiskGuard
API
Parser
Log Entry Handlers
Data Structures
Rules
Limits
API
Parser
Limit Types
Data Structures
Rules
Verification
Accountant
Verifier
Rules
Data Structures
Ledger
Ledger Container
Ledger
Ledger Entry
...
Ledger Values
...
Ledger Variables
...
Ticker Tape
Ticker Tape Container
Ticker Tapes
Ticker Tape
Tape Entry
...
Tape Helper
...

7.2. Language

For the language, we decided to use Python, because this is what developers at Optiver are familiar with. Also, because we decided to make a prototype first, Python is a language with a very high development speed. In the research report, we already talked about doing the whole project in Python, but we had doubts about the speed of the language. After two weeks we had finished the prototype and ran the first tests. As we would see later, this prototype was still missing a lot, but it did contain a solid architecture and was easily extendible to add the missing features. We considered to switch to another language with better performance than Python at this point and finish the project in that language. The trade-off would then be getting better performance while giving up about two weeks of work. We prioritized getting to a working product in the 10 weeks over making it as fast as possible, and we decided to continue developing the existing project in Python.

7.3. Usage of a Framework

In our research report, we made clear that we thought that Python would be too slow when running over the data files. As a solution, we researched frameworks like Flink and Spark, which enable you to set up parallelized and/or clustered computing solutions with relatively little effort. In the end, we decided that we wanted to try and use Flink if needed over other frameworks because it a) has the best Python interface, b) provides theoretically the biggest speedup compared with its alternatives and c) provides a no-failure guarantee by redistributing the work of failed nodes to other nodes. For more information about our choice for Flink, see Appendix E.

For our prototype, we first implemented a version without the usage of Flink to increase the development speed. Because this appeared to be too slow (i.e. at least 500 hours to analyze a single day as opposed to the requirement of no more than 8 hours), we re-evaluated the options to improve the performance about which you can read more in the section 9.2 Parallelization. To summarize that section, we decided that we would omit the usage of Flink based on our estimates of the performance of just utilizing all cores on a single machine and based on the preferences of our client Optiver. This choice is motivated by noting that analyzing a single day in 8 hours on a regular machine (which is achievable by utilizing all cores simultaneously) is fast enough for the day-to-day analysis desired by Optiver as stated under 2. Problem Definition. Additionally, Flink requires a big cluster of computers to provide a significant advantage, which is a problem because Optiver a) does not currently have a setup to provide such a cluster to Flink, b) they are very reluctant to outsource this operation to third-parties because the information is very confidential, and c) our program is not a critical part of their infrastructure, so the fault-tolerance provided by Flink does not have a sufficiently high priority.

7.4. Storing Data

In Python 3.6 there are 4 usual ways of storing data: tuples, classes, namedtuples and dictionaries. We are using all 4 of them because all of them have their own set of advantages that make them suitable for specific situations:

- **Tuples** are used to store data when we know the fields in advance, (almost) all of the fields are used, and we need the very best performance we can get. An example is storage of order types that can be skipped, which are a composition of several fields of the order.
- **Classes** are used to store data when we know the fields in advance, (almost) all of these fields are used, and we need very fast access to the fields. An example is the LimitData class that is an aggregation of all limits active at a certain point in time. An additional advantage of classes is that they allow us to add type hints for the parameters and that they are easily extendable when we would like to add some functionality to the classes.
- **Namedtuples** are used to store data when we know the fields in advance, (almost) all of these fields are used, and we don't need fast access to the fields. They greatly reduce the amount of boilerplate code compared to classes. An example is the Breach namedtuple.
- **Dictionaries** are used to store data when there are a lot of fields and some of these fields could be unused. An example is the dictionary for an order which can have 34 fields of which only 8 to 12 are

used in most cases. Because dictionaries need to hash the key for every access, which is a relatively slow operation, we mapped the field names to integers in the performance-critical case of the order dictionary and used those as keys. This eliminates the hashing overhead, but adds extra boilerplate code and makes the debugging harder.

8. Implementation

In the section 3. Problem Analysis the input and output files are listed. Essentially, our program is a transformation process that transforms the input to an output. Figure 3 shows an overview of the implementation of the program, with below some explanation of the main parts.

- The **Accountant** is responsible for managing the entire process and does not contain a lot of logic. It just calls in the correct order the functions exposed by the other modules and can be regarded as an implementation of the Façade design pattern. The Order Log modules parse and yield the orders one-by-one
- The **Audit Record** modules parse the audit records and provide a mapping from (feedcode, market) to (underlying product, multiplier, kind, currency)
- The **RiskGuard** modules parse the RiskGuard logs and provide a mapping from (currency, time) to the exchange rate of the currency and from (market instance, user, market, underlying, time) to (role, trading type)
- The **Limit** modules parse the limit history database and provide a mapping from (market, underlying product, trading type, role, time) to all limits that are present at that moment
- The **Ledger Container** contains all the ledgers that acts as the order books which keep track of all outstanding orders. This module is explained in more detail in the section 8.1 Ledger
- The **Ticker Tape Container** contains all ticker tapes that keep track of all activity within a certain time frame. This module is explained in more detail in the section "Ticker Tape"
- The **Verifier** solves our problem by comparing the values returned by the ledgers and the ticker tapes to the current applicable limits. Essentially all other modules only exist to give the Verifier all relevant data.

8.1. Ledger

The Ledger Container class is the class owned by the accountant and contains several ledgers, which are our version of order books. It maps every *market, underlying product, user* combination to a ledger.

Ledger: applicable limits:

The Ledger imposes limits on all outstanding orders. These limits are:

- *Max number of outstanding orders*
- *Max gross volume of outstanding orders*
- *Max gross value of outstanding orders*
- *Max outstanding volume per instrument*

Because the way to calculate the values for these limits is different for every one of them, the ledger has 4 variables, one for each limit, that are updated every time the order book changes. Figure 4 and 5 illustrate an extensive visualization of the workings of the ledger.

This section caused significant issues throughout the project. The first problems arose when it was indicated that there was no guarantee on time safety for the replies from the market. This provides some edge cases in which orders and the market behave in very exotic manners. Furthermore, the replies given by a market were inconsistent. These issues varied from surprising, but not completely unexpected, to puzzling. One of the not completely unexpected issues was caused by the inconsistency in receiving the same kind of replies from the market due to different implementations at the exchanges. Alternatively, some of the more puzzling issues range from being unable to track orders due to a bug in the tracker id of the order, to not receiving any replies from the market at all. All in all, these issues consumed a lot of development time. For most of these issues, one can reflect that some intuitive, though erroneous, assumptions were made in the process. This can mostly be attributed to our personal unfamiliarity with the

data. This unfamiliarity surfaced in the form of spending frustrating hours on finding and resolving errors in our expectations that were masquerading in our perceptions as bugs while being oblivious to the underlying cause.

8.2. Ticker Tape

The Ticker Tape Container class is the class owned by the accountant and contains several collections of ticker tapes. It maps every *market, underlying product, user* and every *market, feedcode, user* combination to such a collection. Figure 6 illustrates both the structure of the ticker tape classes and the call hierarchy when the ticker tape is updated with a new order.

Ticker Tape: applicable limits

The Ticker Tape manages all limits that check the activity per time interval. These limits can be segmented into 2 categories:

- Limits checked per underlying, mapped per *market, underlying product, user*
  - *Max number of quote operations*
  - *Max number of insert / amend operations*
  - *Max gross traded volume*
  - *Max gross traded volume over a long term*
  - *Max gross traded value*
- Limits checked per instrument, mapped per *market, feedcode, user*
  - *Max number of quote operations*
  - *Max number of insert and amend operations*
  - *Max gross traded volume*

To check these limits, we store on a ticker tape (which is implemented as an ordinary double-ended queue) every time a new order arrives the order and remove all orders that are outdated, i.e. fall outside of the window size. We're using a MapReduce based scheme for the Ticker Tape implementation:

- To improve performance, we only store the relevant data on the tape by mapping the order attributes to a tape item.
- To check the limits, we calculate an aggregated value from all values stored on the tape by reducing the tape items so that we can compare this value with the limit.

We can segment the limits into 3 groups that use different mappings and reductions:

- Maximum amount of operations per time interval
  - *Maximum number of quote operations*
  - *Maximum number of insert / amend operations*
  - *Maximum number of quote operations*
  - *Maximum number of insert / amend operations*
  - **Corresponding mapping**: order → timestamp
  - **Corresponding reduction**: number of tape items
- Maximum volume per time interval
  - *Maximum gross traded volume*
  - *Maximum gross traded volume over a long term*
  - *Maximum gross traded volume*
  - **Corresponding mapping**: order → timestamp and volume tuple
  - **Corresponding reduction**: sum of volumes
- Maximum value per time interval
  - *Maximum gross traded value*
  - **Corresponding mapping**: order → timestamp and value tuple
  - **Corresponding reduction**: sum of values

Note that we are always storing the timestamp on the tape because we need the timestamp to know when an order is outdated, i.e. the time between this order and the newest order is larger than the window size.

Figure 3

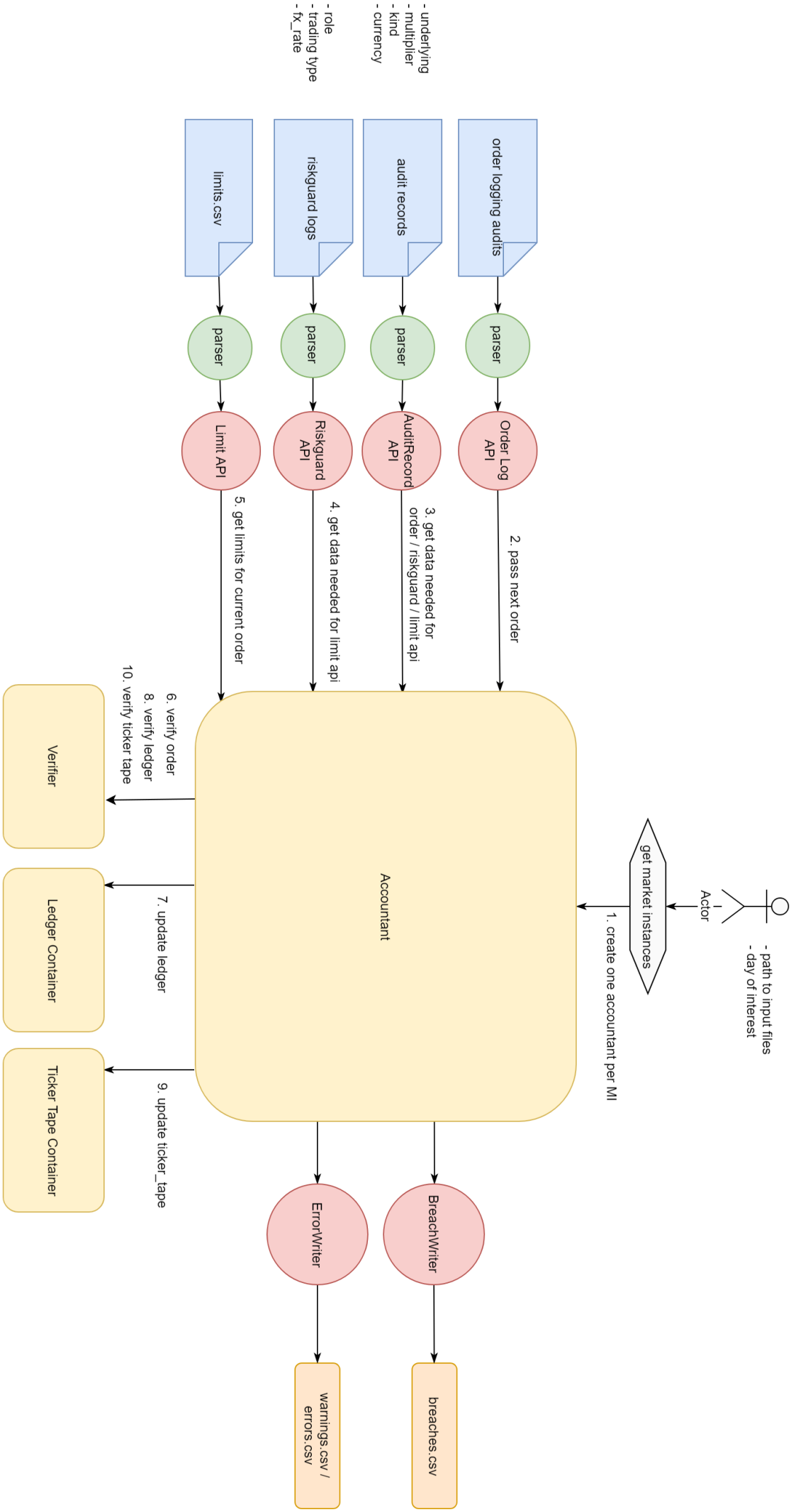

- path to input files
- day of interest
Actor
get market instances
1. create one accountant per MI
Accountant
order logging audits
audit records
riskguard logs
limits.csv
- underlying
- multiplier
- kind
- currency
- role
- trading type
- fx_rate
parser
parser
parser
parser
Order Log API
AuditRecord API
Riskguard API
Limit API
2. pass next order
3. get data needed for order / riskguard / limit api
4. get data needed for limit api
5. get limits for current order
6. verify order
8. verify ledger
10. verify ticker tape
7. update ledger
9. update ticker_tape
Verifier
Ledger Container
Ticker Tape Container
BreachWriter
ErrorWriter
breaches.csv
warnings.csv / errors.csv

Figure 4

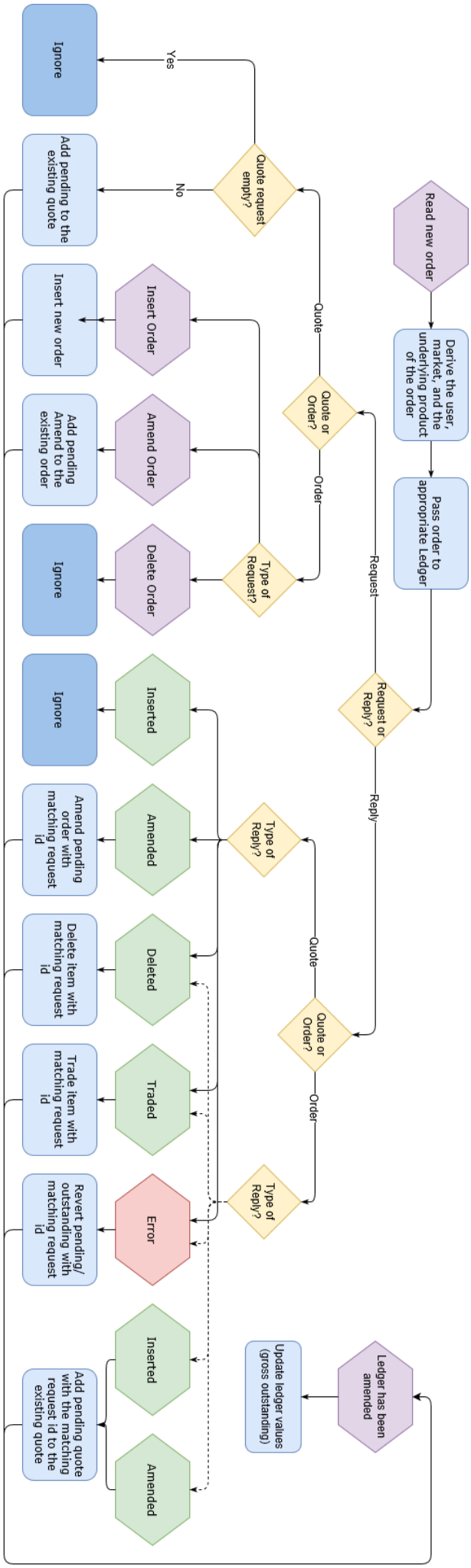

Read new order
Derive the user, market, and the underlying product of the order
Pass order to appropriate Ledger
Request or Reply?
Request
Reply
Quote or Order?
Quote
Order
Quote request empty?
Yes
No
Ignore
Add pending to the existing quote
Insert Order
Amend Order
Delete Order
Insert new order
Add pending Amend to the existing order
Ignore
Type of Request?
Type of Reply?
Inserted
Amended
Deleted
Traded
Error
Ignore
Amend pending order with matching request id
Delete item with matching request id
Trade item with matching request id
Revert pending/ outstanding with matching request id
Quote or Order?
Quote
Order
Type of Reply?
Inserted
Amended
Add pending quote with the matching request id to the existing quote
Update ledger values (gross outstanding)
Ledger has been amended

Figure 5

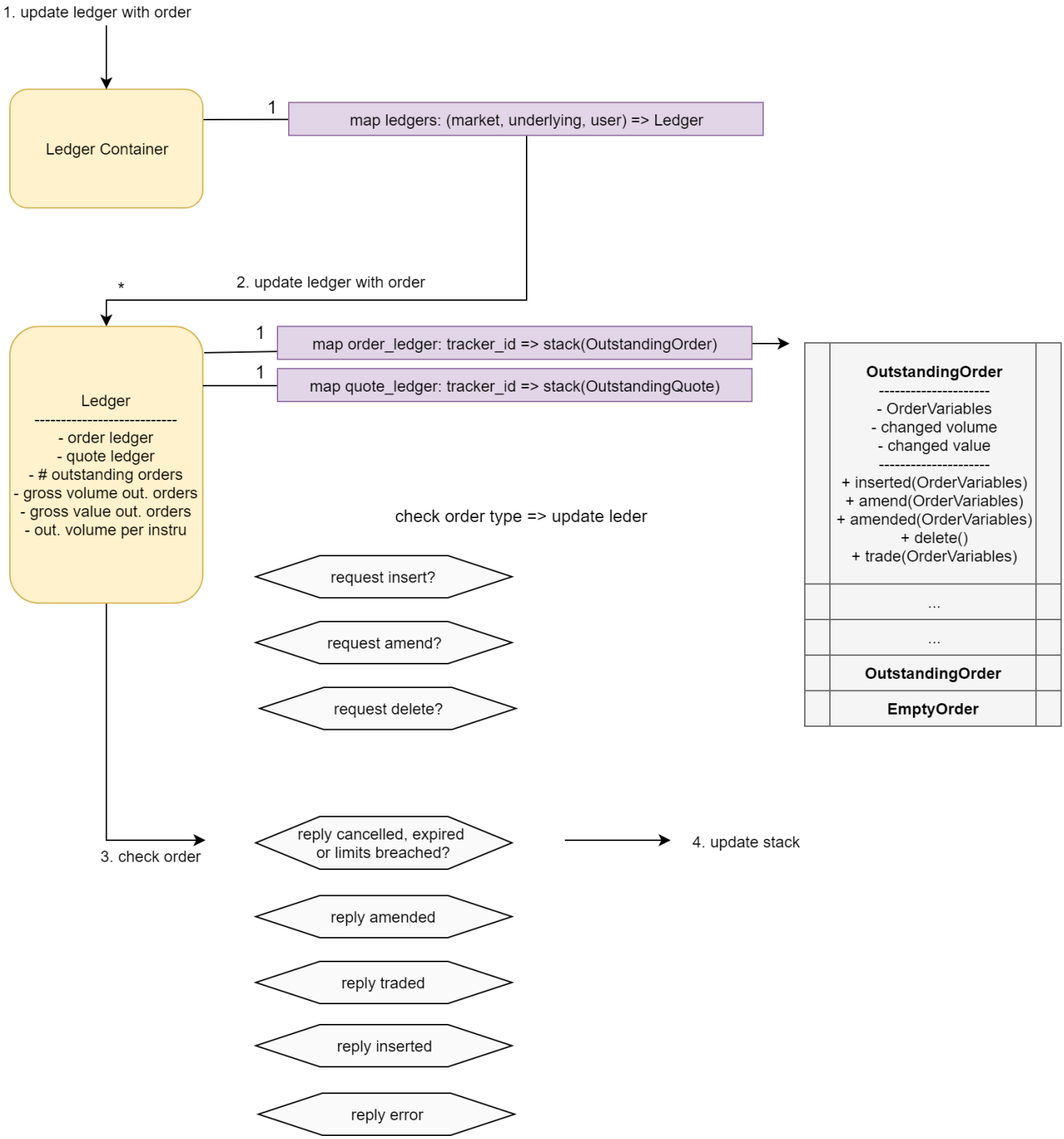
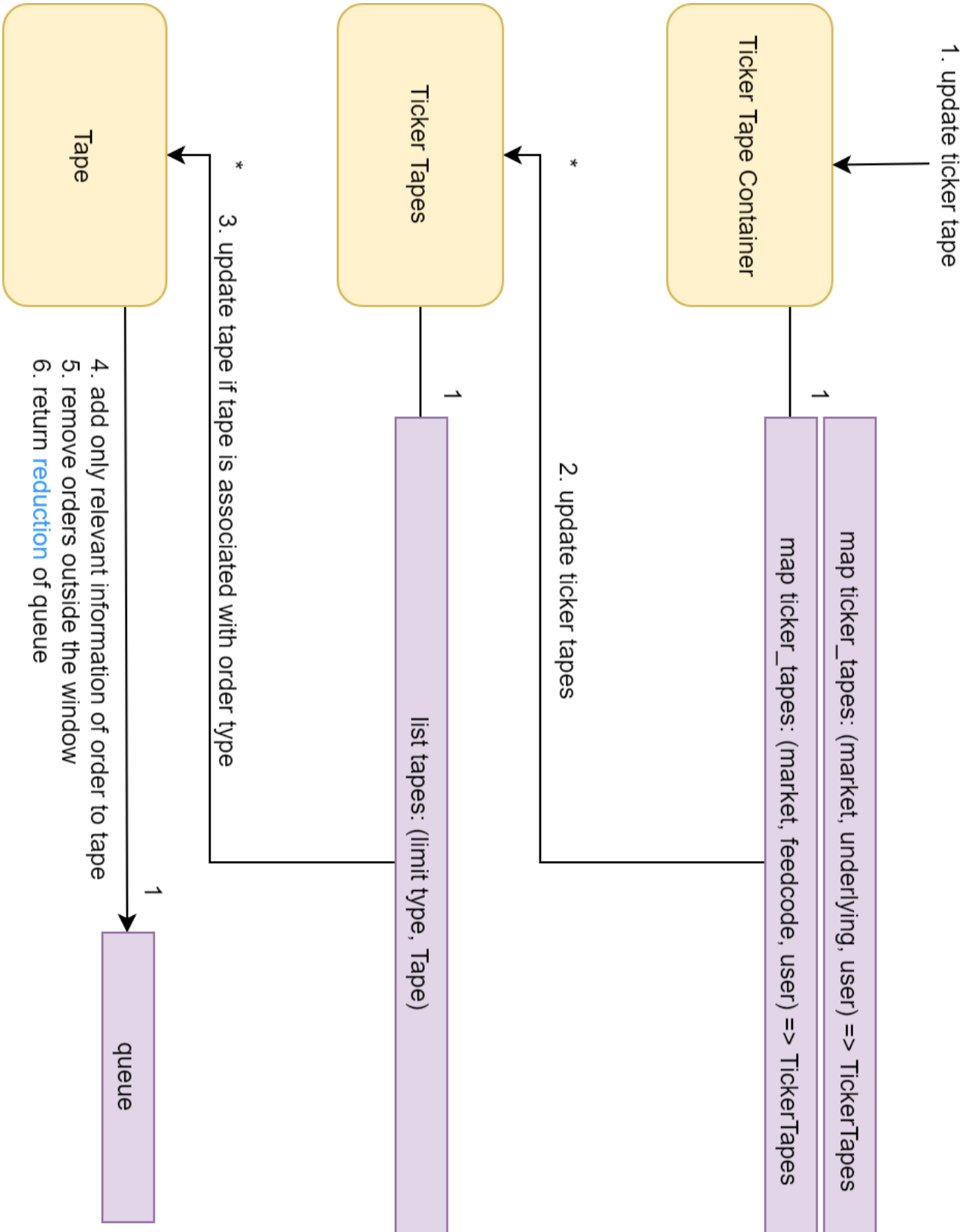

1. update ledger with order
Ledger Container
1
map ledgers: (market, underlying, user) => Ledger
*
2. update ledger with order
Ledger
- order ledger
- quote ledger
- # outstanding orders
- gross volume out. orders
- gross value out. orders
- out. volume per instru
1
map order_ledger: tracker_id => stack(OutstandingOrder)
1
map quote_ledger: tracker_id => stack(OutstandingQuote)
OutstandingOrder
- OrderVariables
- changed volume
- changed value
+ inserted(OrderVariables)
+ amend(OrderVariables)
+ amended(OrderVariables)
+ delete()
+ trade(OrderVariables)
...
...
OutstandingOrder
EmptyOrder
check order type => update leder
request insert?
request amend?
request delete?
3. check order
reply cancelled, expired or limits breached?
4. update stack
reply amended
reply traded
reply inserted
reply error

Figure 6


1. update ticker tape
Ticker Tape Container
1
map ticker_tapes: (market, underlying, user) => TickerTapes
map ticker_tapes: (market, feedcode, user) => TickerTapes
*
2. update ticker tapes
Ticker Tapes
1
list tapes: (limit type, Tape)
*
3. update tape if tape is associated with order type
Tape
1
queue
4. add only relevant information of order to tape
5. remove orders outside the window
6. return reduction of queue

9. Speed optimization

Because the proof of concept version of the program takes 500 hours to analyze a single day on a single thread and the client requires that the program can analyze a single day within 8 hours, the performance had to be increased. There are several ways we considered to improve the performance:

Run Time	Total	Largest Market Instance
Proof of concept	500h	25h

9.1. Profiling

We focused on increasing the efficiency of the existing code by profiling the program and analyzing which parts were seriously degrading the performance of the system. For this, we used both cProfile and the built-in profiling functionality of PyCharm and reduced the original execution time from approximately 500 hours to 50 hours for a single day. However, we must keep in mind that we also changed several non-performance related things in the program that may have impacted the performance (e.g. ignoring some irrelevant order types).

Run Time	Total	Largest Market Instance
Proof of concept	500h	25h
Post optimisation	50h	2.5h

Examples of the optimizations that we performed include:

- Replacing the built-in Python functions for parsing a string representing a date/time by a hard-coded variant (e.g. characters 10-11 encode the month, 13-14 encode the day)
- Replacing dictionaries by tuples. Unfortunately, the result was harder to debug because you have to remember which element in the tuple corresponds to a certain value. However, we created some mappings (in the form of constants) so that we do not use arbitrary numbers but instead a variable and we justified the usage of tuples by a significant boost in performance because the time spent on hashing was reduced to 0.
- Replacing multiple references of the same variable by a single reference of which the result is stored in a variable. For example, if a method executes several times *order[VALUE]*, we created a local variable that stored this value and use this local variable instead. The impact is generally negligible, but not when a very small piece of code is executed trillions of times.
- Updating the window sizes for the ticker tape only when the limit sizes have changed
- Using pre-calculated hash values for immutable objects
- Opening the output files only once instead of every single time the files are appended with a string
- Return an entire array at once instead of yielding the array element-by-element
- Calling a helper method (e.g. *is_followup_order()*) only once and store the result

9.2. Parallelization

Running the program on multiple cores on the same CPU is a relatively easy and very cheap way to boost the performance when the program is easily parallelizable (which it is because every day and every market instance per day is entirely independent of the others). This system creates a process-pool that runs the program on a certain number of processes as specified by the user. The program uses a greedy algorithm to analyze the log files in an efficient way: at every iteration, every thread takes the biggest file that is not

(being) evaluated yet by another thread and analyzes the entire file before continuing to the next file. This algorithm gives a close-to-optimal result, which is good enough for Optiver.

Run Time	Total	Largest Market Instance
Proof of concept	500h	25h
Post optimisation	50h	2.5h
Post Parallelization	6h	2.5h

9.3. Operating System

The program was developed on machines running Windows 10. Because Windows might be relatively slow compared to other operating systems, we also tested the program on a Linux machine. We got access to a Linux machine running CentOS and with a CPU that is approximately 20% faster than the CPU of our development machines. The results were as follows:

Run Time	Total	Largest Market Instance
Windows 10	6h	2.5h
CentOS Linux	5h	2h

The difference between the operating systems seems to be quite small because the performance was 20% higher on the Linux machine, but the CPU was also 20% more powerful on the Linux machine. The final program can be deployed with just a tiny modification in an OS call on Linux machines, but we don't necessarily recommend doing it.
The specifications of the machines used are:

- Windows Machine
  - Operating System: Windows 10 Enterprise 64-bit
  - Processor: Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz (4 cores) (Passmark CPU Score = 9701)
  - Memory: 8192MB RAM
- Linux Machine
  - Operating System: CentOS Linux
  - Processor: Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (4 cores) (Passmark CPU Score = 12209)
  - Memory: 7823MB

For further reference, due to early unavailability of the deployment machine, time-sensitive approximations are defined as per the development machine.

9.4. Interpreter

The default Python interpreter is called CPython. CPython is an interpreter written in C and compiles the Python code into bytecode before interpreting that bytecode in an evaluation loop. Because it interprets the bytecode it is not as fast as other languages like C (see https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/python3-gcc.html). There are several other Python interpreters/compilers, most of which let Python run on other virtual machines (like the Java Virtual Machine or the Microsoft CLR). There are 2 common implementations optimized for speed. We evaluated both of them:

- **PyPy** consists of a translation framework that translates Python code to RPython, a subset of normal Python, and of an interpreter. It separates out all platform-independent parts of the program from the things that can be optimized for a specific platform and adds things like Garbage Collection (GC) and Just-In-Time compilation (JIT). Because PyPy is still only available for Python 3.5 or older we had to convert some Python 3.6 features like enums to make the code compatible with PyPy. PyPy did not increase the performance while parsing the input files (it even decreased the performance). This is most likely due to the fact that loading the input files is I/O-bound instead of CPU-bound. On our development machines we got the following loading times:
    - Processing the limits takes 22 seconds with CPython, 47 seconds with PyPy
    - Processing the riskguard files of a random day (we took February 27, 2018) takes 25 seconds with CPython, 67 seconds with PyPy
    - Processing the audit records of a random day (we took February 27, 2018 again) takes 29 seconds with CPython, 29 seconds with PyPy
    - Processing an entire day on a single thread takes 50 hours with CPython, 30 hours with PyPy
    - Unfortunately, the multiprocessing module in PyPy is dysfunctional (in Windows) due to a call in the internal PyPy libraries to a non-existent function
        - A raw, self-developed, implementation of a multiprocessing system allowed for an increase of about 8%

Run Time	Total	Largest Market Instance
CPython Single Process	50h	2.5h
CPython Multiprocess	6h	2.5h
PyPy Single Process	30h	1.5h
PyPy Multiprocess	5.5h	1.5h

        - However, due to the increased unreliability because of the raw implementation of a multiprocessing implementation, we decided to retain a preference for the CPython interpreter until a more resilient solution exists.
- **Cython** is a compiler that compiles Cython code, a superset of normal Python, to C modules that can be either generic C libraries or libraries built specifically to work with Python. Just running our regular code with Cython does not result in a performance increase, because Cython only optimizes parts of the code that are specifically written for Cython using a special syntax. We ran several tests and rewrote performance critical sections in Cython syntax only to discover that the overhead of calling these C functions from the Python code was quite big. The solution is to rewrite the entire handling of an order logline in Cython syntax for which we did not have enough time because we did not know if the speed-up would be worth the time.

10. Software Quality & Testing

One of the most important requirements of the code for our client Optiver is that the code has to be so-called "clean code" to prevent bugs from happening and make it easy to maintain the software in the future. An elaboration on every characteristic of software quality can be found below:
These characteristics of software quality are as explained in the book "Code Complete, Second Edition" (McConell, 2004).
Please refer to 15. Discussion & Recommendations for our recommendations on how to improve upon these points

10.1. Software Quality

Usability: the program is a command-line tool and can be configured by using several parameters. The output of the program is stored in 3 log files which are grouped in "warnings", "errors" and "breaches". The program is not designed to be so user-friendly that it can be used by normal consumers, but by employees working at Optiver who are quite familiar with command-line tools. Optiver explicitly made clear that a building a GUI around the application has a very low priority for them and is something that should be done only when the program itself is finished.

Efficiency: the program satisfies the constraints imposed by a secondary problem (see "Problem Definition") in the sense that it can analyze a full day of data within 8 hours. However, because the program is interpreted instead of compiled, the performance is lower than it could have been.

Reliability: the reliability of the program is a mixed bag. On one hand, the reliability of the program is extremely high in the sense that every exception that occurs in the program is caught, logged and the program continues with the next iteration. On the other hand, the program is designed to run on a single machine, so if that machine fails, the entire program has to be restarted. We do have to nuance this statement because the days are evaluated in linear order, so if the program crashes after evaluating half a year, it can be configured to simply continue where it left off when the program is restarted. Additionally, the program is tailor-made for our client Optiver that uses exclusively Intel Xeon CPUs (which have a lifespan that is considerably above regular consumer CPUs) and has several emergency electricity generators in case there is a power outage. This means that the chance that the server on which the program is running fails is comparatively small.

Integrity: to reduce the chance that the program is abused to read or write to files that it should not read or write to, the program never asks for more permissions that it needs to. For example, it does not ask for writing permissions to read the input files. Additionally, to ensure that the data is accessed properly, the program does not use hardcoded strings anywhere, but either pre-defined constants or enumerations.

Adaptability: the extent to which the system can be used, without modification, in applications or environments other than those for which it was designed (i.e. Windows 10) is good. To illustrate this, the program was run on Linux (CentOS) which was possible with only minor modifications. The reason for this is that the system is built with Python which has good portability in general.

Maintainability: to keep the ease with which you can modify the system to change or add capabilities as big as possible, we made several decisions to accommodate this goal

- The system is written entirely in Python, which is a very flexible language and allows for easy maintenance of the system. However, we do have to consider that Python can also be easily abused resulting in a lot of "hacks" which make in the long-run the code unmaintainable.
- The system adheres as much as possible the Single Responsibility Principle (SRP). As a result, the system is highly modular which makes it easy to add or change capabilities because the coupling between the modules is quite loose. There are no singletons or global objects, but all data a class needs is dependency injected which makes the code both quite predictable and easier to maintain.
- The system follows the Law of Demeter everywhere which simply means that every class knows only its direct dependencies. This ensures that object containers can be changed without having to rework their callers.
- All constants are abstracted away from the source code and the data flow is separated from the error flow, both of which make it easier to change the source code without breaking unrelated parts of the program.

Flexibility/portability: we did not design the system so that it can be used in other environments other than Optiver, because it is extremely unlikely that anyone else will use the program. Nevertheless, it is still reasonably easy to change the program when the environment changes, see Maintainability.

Reusability: to promote the extent to which and the ease with which parts of the system can be used in other systems, we separated all code that is specific to our system (i.e. non-reusable code) from all code

that could also be used in other systems (i.e. reusable code). This separation is illustrated in the section 7. Design:

- **Data Structures** which contains data structures, e.g. classes, that are communicated between modules
- **Rules** which contains the immutable data such as constants, definitions and rules which are not specific for our program
- **Utils** which contains shared functionality across classes and which is not specific for our program, e.g. the conversion of a timestamp to a `date.time` object or the extraction of the name of a market instance from the name of a corresponding log file
- **Files** which contains all file-system related functionality which is reusable across many systems

Readability: we put a lot of effort into keeping the codebase readable and in the style that Optiver prefers, namely the style dictated by the book Clean Code (Martin, 2008). This means that we paid attention to several things to keep our code readable

- The code is self-documenting, i.e. every function that was considered complex is split up into multiple smaller functions with a descriptive name. The Accountant class is an excellent example of this.
- When methods are already quite short but still hard to understand, there are always comments to help the reader understand the code.
- The usage of if/else statement is kept to a minimum by using polymorphism. For example, all order and quote related classes use this principle so that their owners do not have to check if the type of the order is a "quote" or an "RFQ".
- The usage of negative conditionals is reduced to a minimum because it is much easier for humans to comprehend positive reasoning.

Testability: because encapsulation is impossible in Python, it is quite easy to test everything. However, to reduce the need to create a new instance of the used objects for every single test, we keep the number of fields of every class to a minimum and prefer the "functional programming" style where the data is passed around. Additionally, to reduce the need to know how the tested classes work internally (i.e. regard them as a black box which is desirable when testing the public interface), all variables are dependency-injected which allows us to use Mocks instead of real objects. In the tests, we're exclusively using Mocks instead of real objects to guarantee that if class A changes, the tests for class B (that might be using class A) do not break.

Understandability: there are several features that make the system more comprehensible as a whole:

- The system is segmented into several modules and files which are consistent in their naming across all modules. As illustrated under 7. Design, all API's are named "API", all parsers are named "parser" and all reusable code is split up in "data structures", "rules", "utils" and all classes within the "files" module.
- The classes use as few fields as possible and instead pass the data around adhering to the functional programming style. This allows us to build many methods without side effects which are both easy to understand and to test.
- In Python encapsulation is impossible, but PyCharm can "hide" variables and methods when they're prefixed with an underscore. We used this extensively on everything that should be private. This greatly reduced the size of the public interface of the classes making them easier to understand.

Documentation: we added documentation for every public method that gives an overview of what the method does. In most cases, we did not add documentation for the parameters as discussed with the client, because this is generally not needed to be able to understand the function of the parameter and would add a lot of clutter to our code. We used comments in our code, but sparingly, because our client explicitly stated that they prefer to subdivide a method into a few smaller methods with very clear names because the code should be self-documenting.

Encapsulation: in Python, it is impossible to completely encapsulate a field, method or class, but PyCharm can "hide" these entities when referenced from the outer scope when their name is prepended by an underscore. We used this trick extensively and applied it to every entity that would be private in a language with support for encapsulation. The reason we do so is that it greatly reduces the dependencies on "details"

which makes the code more flexible and easier to maintain. Additionally, this encapsulation helps to test the code and maintain those tests. Because we must tests the full interface of every class to be confident that the class works correctly, we'd have to test a lot more when everything is exposed (as is the case without encapsulation).

Robustness: generally, it is important that a software system continues to function in the presence of invalid inputs. This can be done by using defensive programming and by adding pre-conditions, post-conditions or class-invariants. We considered adding these features to our program but decided not to do so, resulting in a non-robust program. We made this decision because the overhead of validating the input does not really make sense, because the input is not user input that can be messy, but loglines which are generated by a computer and which are represented in a format that adheres to very strict rules. Additionally, because the number of orders that the system will analyze will easily be in the range of 100s of billions of orders, the total extra time spent on validating every single order will be quite large and is not worth it.

10.2. Test

Accuracy: to test how well the system does the job it is built for, namely verifying that the limits imposed on the orders are not breached, we wrote documentation tests that test for every line of documentation if the product behaves according to the documentation, and integration tests that test with a lot of sample scenarios if the output of the program corresponding to the input of the program is correct.

- **Documentation tests:** the system is tested with almost 200 documentation tests (more will be added in the last week) that test relevant lines of the (testable) documentation. This is done quite thoroughly by exhaustively testing all combinations when there is a limited set of possible user inputs and verifying the output of every scenario is correct. For example, every risk limit has a line of documentation that describes to which order types and instrument types it applies, and every line has 12 corresponding tests that test all possible combinations.
- **Scenario tests:** the scenario tests run on a single scenario, which consists of a small number of orders that emulate an interesting situation and of a ground truth that maps every order to the expected state of the ledger or ticker tape. All the scenarios are copy-pasted from the real log files to make sure that the tests resemble reality as much as possible. Every time we run the program on the real data and we found a bug in our program after analyzing the results, we added the loglines responsible for triggering the bug as a scenario to verify that the bugs were fixed and stayed fixed.

Correctness: to be confident that our program does not contain faults in its specification, design and implementation we added several types of tests. The test cases cover a total of 94% of all code, see Appendix H, and can be further attributed to the following categories:

- **Unit tests:** every file has a corresponding file containing all unit tests for that particular file. The unit tests test the functionality of every single method and verify that it works correctly. We did this very thoroughly with a lot of bad-weather scenarios, tests for exceptions and very extensive use of mock-objects to reduce the dependency on other code. We have a line coverage of 94% and strive to get it up to 100% in the last week, not by just running every line, but by testing all edge cases without looking at the code.
    - We tested methods that return a Boolean value (i.e. true or false) in a slightly different way than the rest of the methods. Their return values can be classified as true positive (TP), false positive (FP), true negative (TN), and false negative (FN). Every Boolean method has at least one TP and at least one TN test, but generally more than one so that we can be more confident that we will never get an FP or FN.
- **Integration tests:** to verify that the verification module - the main module of the program - works as expected, we wrote several extensive integration tests. These tests create an accountant in the same way as happens in release mode (so including the creation of several ledgers, ticker tapes, etc.), repeatedly giving the accountant a new logline to process, verifying if the entire state of the entire ledger and/or ticker tape matches our expectations. This is done by inspecting the private variables, which is not always a good idea when testing a program. However, in this case, it is justified because practically all methods are private (and have no reason to be public other than for

testing purposes) and the amount of logic needed to verify the orders is huge. Therefore, we want to ensure that the black box within the verification module behaves correctly - which is achieved by running these integration tests.

11. Results

The goal of the project was mainly to give the guarantee that the limits are not breached by the orders placed by the market links. Additionally, part of the project was to run the program on some historical data and pass the results of the analysis to the risk department at Optiver.

The report that you are reading had to be turned in 1 week before the end date. Because of some unexpected edge cases we had to solve and some bugs we found one week before, we did not have time to evaluate all the data we got. The last week of this project will be dedicated to finishing the program and also to evaluating more of the data. If any extra interesting results are found, these will be presented at the final presentation. For now, one breach has been found, but this one was already found by the team at Optiver. Below is a description of the breach.

11.1. Safe amends at the Brazilian exchange Bovespa

Quite often we found breaches on the exchange Bovespa. These breaches were all related to the outstanding orders limits, namely *max_gross_outstanding_value* and *max_gross_outstanding_volume*. The breaches were quite small, only exceeding the limits by a couple of percentage points, but we found these on almost all days that we tested. An example of the breaches is shown in Appendix C.

After manually checking the contents of the ledger this breached appeared to be a real breach. While talking this over with the risk team, the question was raised if we were taking safe amending into account. When amending an order with volume 10 to a volume of 15, and getting a trade for 10 in between, normal exchanges will still insert a volume of 15. This gives a higher maximum executable volume, and safe amending ensures that at the example from 10 to 15, with a trade of 10 in between, the resulting volume is 5. We took the conservative approach of always adding the volume of the outstanding order to the amount of the amend order. This was told to us as being the approach the current RiskGuard is taking. But for the Bovespa exchange, the market link already has implemented the safe amending in the risk limits. It is currently also being implemented by the developers in the RiskGuard, and we should also change our program accordingly.

12. Evaluation Product

Our problem was that we needed to find a way to verify these order logs against the limits in a reasonable time frame. Our value proposition is (summarized)

- Check if limits where breached on a past date
- Check the limits in a reasonable amount of time
- Inspect the log files and report all inconsistencies

We added value to the company by working on these proposals and achieved the following things:

Check if limits where breached on a past date

- The results showed the case of breaches on the Brazilian exchange which occured because of safe amends, which were not implemented in their own risk system yet. As this is not a new discovery, it does not add a lot of value. It does show that our verification system works.
  Because we got not a lot of results, the value we added is mostly in giving Optiver more confidence about the correctness of their trading algorithms and risk systems.

Check the limits in a reasonable amount of time

- We did this by extensive profiling, identifying the parts of the program that cost most time and optimizing those parts. A comprehensive overview of all issues we found and optimized can be found in the section "Speed Optimization".

Inspect the log files and report all inconsistencies

- To create integration tests that recreated the reality, we inspected the log files and copied interesting scenarios. We also inspected the log files when our program reported a warning, error or a breach to debug the issue. While doing this we noted several inconsistencies in the data that we reported to the developers. All inconsistencies we found in the log files are listed below, and are further explained in the section 13. Additional added value:
    - In a Traded reply the volume does not mean the same for RFQs
    - Missing Expiration field
    - Missing price and volume field on request, but not on reply
    - Faulty order types
    - Missing replies

12.1. SIG

The TU Delft provides access to an evaluation of the software product by the Software Improvement Group (SIG), a company specialized in evaluating software repositories and grading them on an industry-based scale. SIG performed a manual evaluation on our code base in weeks 6 and 9 of the project. The evaluation performed in week 6 provided us with feedback on what we could improve in our program. We got the opportunity to use this feedback and apply their recommendations. In week 9 we handed in our updated code base again so that SIG could reevaluate the program and their points for improvement.

The feedback

6 workdays after we handed in the code at SIG we got the feedback (full text can be found in Appendix G). It was given a rating of 3.4 out of 5, which indicates that it is maintainable according to the market average. Two units of improvement, namely interfacing and complexity, were given as focus points for improvements.

Interfacing: SIG gave the FieldDefinitions class as an example because it had a high number of constructor parameters. We had copied and slightly modified this piece of code (because we migrated from Python 2 to Python 3) from an internal software system at Optiver. Nevertheless, we took the feedback to heart and refactored it into a group of similar classes with significantly fewer parameters, either 1 or 2 per class (see `src/mi_log/rules.py`). The example given was not the only case of issues with regards to interfacing, hence we thoroughly reviewed the code base for interfacing issues. Some notable examples are:

- The addition of LedgerVariables in `src/verification/order_types/ledger_variables.py`, which previously where all separate arguments for the classes in `src/verification/order_types/ledger_entry.py`, `src/verification/order_types/outstanding_order.py`, and `src/verification/order_types/outstanding_quote.py`.
- The addition of AuditRecordsData in `src/auditrecords/data_structures.py`, which previously was a used either as separate parameters or as a tuple with 4 values.
- The improved use of abstract classes in `src/verification/order_types/` for both ledger entries, ledger variables, and ledger values. Hiding the inner workings of these classes behind a common interface derived from the actions seen in the order logs.

We carefully evaluated if it actually made sense to refactor the code to comply with the interfacing standards. One example of where we decided not to make a change to comply with this metric is the method that returns the role and trading type from the RiskGuard API (see `src/riskguard/api.py`). This method requires 5 parameters to find the correct role id and trading type, all of which are required and do not depend on each other in any other way. Because Python has support for keyword arguments, which significantly improve the readability of a function with a large number of parameters, we decided to keep the 5 parameters and to use those keywords arguments. Apart from that, we generally capped the number of parameters per method, function, and, constructor to a maximum of 3.

Unit complexity: SIG gave the verify method in the Accountant class (see `src/verifications/accountant.py`) as an example. This method contained about 8 separate steps denoted by comments. The advice was to split this up into separate methods to improve readability and testability. However, the different steps where not mutually exclusive since they relied on overlapping data coming from the order logs. At first glance, this should not be a problem because all this data is available in a dictionary representing an order, but because speed is a major consideration for our program we could not repeat the process of obtaining the information from the dictionary because this would be too slow. Finally, we made a solution that was both elegant and efficient by adding several extra class variables and modifying the parameters passed to the methods. We prefer to apply a more "functional programming"-like style where the parameters are never modified, and classes don't have a state or side effects, but this solution was the best compromise in our view. Apart from this example we also refactored other parts of the codebase which were quite complex. Most parts had to be designed with performance in mind because they would be executed 100s of billions of times and we ended up with a number of changes to reduce complexity. Some notable examples are:

- The parser for the audit records used to be a big single function which has been split up into 3 separate functions that do not depend on each other.
- The parser for the limits history used to be a big single function with a single helper function. This has been split into 3 helper functions, reducing the cyclomatic complexity from 5 to 1.

Lastly, we got a remark that no tests were present in our upload. However, this was not a valid remark as all test could be found in trivially placed `test/` folder. We communicated this back to SIG with the request to evaluate those as well, but SIG did not reply unfortunately.

13. Additional added value

Besides the verification of the limits, we were also testers of the log files created by Optiver. It is important for these log files to be consistent and correct as they are used for compliance purposes.

In a Traded reply the volume does not mean the same for RFQs
In all normal loglines, the VOLUME field in a reply with change-reason Traded denotes the volume that is still outstanding on the exchange, but for RFQs this denotes the opposite, namely the volume that is actually traded. We notified the developers of this issue and this will be changed.

Missing Expiration field
Some files had loglines with a missing EXPIRATION field which gave an error in our parser. The field is unused, so we just ignore it, but for compliance purposes it should not be empty. The issue is sent to the developers for further investigation.

Missing price and volume field on request, but not on reply
A market instance log contained an entry with an amend order request but without a price and volume, which is strange because changing the price or volume is the whole purpose of an amend order. Shortly afterwards, a corresponding reply is found which does include the price/volume. The issue is taken up with the developer.

Faulty order types
Some market instances contained orders which are logged with a faulty order type. For example, futures are logged as spots and vice versa. These particular instances are sent to the developers and will be researched.

Missing replies
A market instance log contained orders without any form of reply present. Appendix D shows the loglines which are found where the last QuoteRequest doesn't have a reply.

Sent to the developers for these logs, and we will try to look for more examples of this mistake.

14. Conclusion

To conclude, we designed, developed and tested a software system that can analyze a significant volume of data in a time frame of fewer than 8 hours. We delivered the product that meets all requirements to Optiver, our client. These requirements, which are found in the section 2. Problem Definition, are re-stated below:

Primary problems

We need to give Optiver the capabilities to verify the order logs against the limits for:

- *A single day of trading*
- *A range of dates*

Secondary problems

We need to give Optiver:

- *An option to analyze a specific market instance, instrument, underlying, user, limit role, or type (order or quote)*
- *An implementation of the program that can analyze a single day within 8 hours*

As explained in this report, the software provides Optiver with the capabilities to analyze a single day worth of data and can also be used to analyze all historical records of data that Optiver accumulated over the past several years. Therefore, we conclude that the primary problems are solved.

Additionally, the software has an extensive amount of options that can be used to specify exactly which type of data the client wants to analyze, and the software can analyze a full day of data in a reasonable timeframe (~8 hours). Therefore, we conclude that also the secondary problems are solved.
As stated under 5. Value Proposition we did more than requested by also adding value to the client by giving them insight into the state and validity of their logging files and reported several issues with the data which have been fixed since.

The final product is a system built in Python that can be used by Optiver to analyze the log data that they generate internally to see if any of the limits they pose on their traders are breached. This system consists of seven components distributed over five main modules. Four modules are responsible for reading, parsing, and interpreting the data. The remaining module is responsible for combining the data provided by the other modules and verifying that no limits are breached. This module consists of 3 sub-components each of which has a slightly different task, namely: the Accountant that acts as the main component and keeps track of the current order, the Ledger that keeps track of all the outstanding orders, and the Ticker Tape that keeps track of all orders in a certain time frame. By using the insights of the ledger and the ticker tape, the accountant can check if any limits are breached and writes them to a log file with all relevant details.

15. Discussion and Recommendations

As is always the case with projects, there are several things that should have been done differently and that we recommend developers working on this project in the future to keep in mind. These recommendations are subdivided into the categories Product and Process.

15.1. Product

Programming language: our choice of programming language, namely Python, is disputable. It is well known that Python is noticeable slower than lower level languages such as Java, C, and C++. So, if speed is one of our top priorities, then why did we choose for Python? Although we have provided numerous reasons why we have chosen for Python under 7. Design, other developers under different circumstances may, and maybe even should, pick a language that has better performance to achieve fast analysis times. For now, running one day in about 6 hours, on a PC with 4 cores, is enough. But when more speed is needed, changing the development language should be considered.

Cluster processing framework: we decided not to use Flink for our program, but instead to process everything on a single machine by utilizing all available cores. We comprehensively motivated this choice in the section 7. Design, but recommend future developers to reevaluate the premises:

- In our case, analyzing a single day in 6 hours on a regular machine is fast enough for the day-to-day analysis desired by Optiver as stated under 2. Problem Definition. However, when the amount of orders per day grows faster than the speed of the individual machines, Flink might be a better solution.
- Flink requires a big cluster of computers to provide a significant advantage, which is a problem because Optiver a) does not currently have a setup to provide such a cluster to Flink, b) they are very reluctant to outsource this operation to third-parties because the information is very confidential, and c) our program is not a critical part of their infrastructure, so the fault-tolerance provided by Flink does not have a sufficiently high priority. When one of these 3 premises is not valid anymore, for example when Optiver does have a cluster of many machines, then the choice to use Flink must be reassessed.

Real-time analysis: this project is an extra check on top of existing real-time risk management systems. Although the risk management system used in the market links keeps track of what orders are outstanding and what is happening in terms of all operations, our program checks what actually happened via the generated logs. It could be valuable to make a real-time version of our program which checks the logs at the same time as they are generated to reduce the time between a breach and the discovery of the breach.

Architecture:

- Future developers should make sure the different rules are as much separated as possible. In our program, we first started with just one class, a LedgerEntry class, and after struggling to keep the type 'quote' and 'order' apart. Later, we found out about the type 'request for quote' which we tried to implement together in the 'quote' class. This gave a lot of problems and complexity, and we spend too much time trying to solve those complexity problems. So besides separating the rules, making it extensible such that when new order types are added at the risk department, they are easily implemented in the program. We did this in the end, but it should have been done from the beginning.
- A second architecture consideration is designing for the presence of different rules for different market instances. These include the presence of safe amends in Brazilian markets and the regarding of orders with the type ZBRQ or TREU as exclusively RFQs. The most important property is the reply structure of trades which has some unique properties. Firstly, depending on the market instance one will or will not receive a cancellation reply after a full trade. Furthermore, in RFQs the traded field currently represents the traded volume, whereas with regular orders and quotes this is the still outstanding volume. Lastly, for FAKs it is possible to receive either a reply for a partial trade or multiple partial trades, ending with or without a cancelled reply

Human-Computer Interaction (HCI): as mentioned under "Software Quality", the program is not designed to be so user-friendly that it can be used by normal consumers, but by employees working at Optiver who are quite familiar with command-line tools. Optiver explicitly made clear that a building a GUI around the application has a very low priority for them and is something that should be done only when the program itself is finished. If that might be the case in the future, then a GUI could provide an intuitive way of configuring the program without having to remember the parameters and make it easy to inspect the result without having to open the generated log files.

15.2. Process

Time estimation

Estimating how long a project will take is always hard, and we underestimated the task of doing this project. The prototype of our program was set up in about 2 weeks, and we thought to be done with implementing all the limits and having a fully optimized version in the 3 weeks after that. In reality, implementing all rules, optimizing maintainability and readability, improving the performance and creating tests took considerably more time than expected. Implementing the rules took most the of time, and in the end the edge cases for RFQs were still not fully implemented. This is one of the edge cases we couldn't have foreseen because these were not documented, but we learned about these issues by talking to the respective developers and will detect these edge cases in an earlier stadium next time. As bachelor students, we made a slight underestimation of the time needed to complete the project because of our current experience with projects and in the future, we will make an estimation that is more accurate.

Software development methodology

The software development methodology we applied, Kanban, is not the only logical choice for a project like this. Although Kanban allowed us to work in a very flexible way, which was needed to accommodate the irregular changes of the project, it might have been useful to have a somewhat better planning for the upcoming weeks. For example, SCRUM allows for this kind of more strategic planning. However, we do not think that Kanban significantly affected our project in a harmful way as both us and our coaches felt like we were working roughly in line with the general planning created at the start of the project.

Major time consumers not included in the report

After we had implemented systems to keep track of outstanding orders and quotes, we discovered that some quotes work a bit different, namely RFQs (or Request For Quotes). By debugging our program repeatedly, communicating back and forth with people at Optiver about the workings of RFQs, we managed to merge functionality for both "normal" quotes and RFQs into a single system. The reason this took a lot of time is twofold. On the one hand the way the log systems for RFQs worked in an unexpected way (even for the people at Optiver). On the other hand, it would have been easier to split the system handling quotes in two, because in hindsight it turned out the two systems were not as much alike as we expected.

16. Ethical Issues

We did not find any ethical issues around our program. Essentially with this project, we created a program to verify if any limits were broken and found inconsistencies with the compliance related logs. The verification of the risk limits will help Optiver with keeping their risks at a low level. If one of the algorithms makes mistakes and this goes unnoticed, then this could potentially harm the financial market. There are examples of algorithms causing a so-called 'flash crash' (Harford, 2012) in which the price of a certain stock drops incredibly in a very short time frame. Helping to prevent this is very valuable. Considering our goal to find inconsistencies in the compliance related logs, this is very important for the regulators because they need to have a perfect view of what happened and at what time. Missing data or inconsistent data could be a problem if that part of the data is essential to finish the analysis of the regulator.

There is some critique on market makers like Optiver and the power that they have with their algorithms. Market makers are able to trade the global markets with algorithms and do not need a large number of people to do that. As a result, the profit made per employee is very high. This is what is happening in companies around the world, where technology is slowly replacing humans to reduce the costs and increase the profits. In the future, there might be taxations on companies relative to how they use algorithms and robots to earn money which could then be spent on providing, for example, universal basic income.

References

Anderson, D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business.* Blue Hole Press.

Harford, T. (2012, August 11). *High-frequency trading and the $440m mistake*. Retrieved from bbc.com: https://www.bbc.com/news/magazine-19214294

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* New Jersey: Prentice Hall PTR.

McConell, S. (2004). *Code Complete.* Redmond, WA: Microsoft Press.

Tudor, D. &. (2006). Using an agile approach in a large, traditional organization. *Agile Conference* (pp. 7-). IEEE.

Appendix A. Info Sheet

General Information

Title of the project: Insight into trading limits
Name of the client organization: Optiver
Date of the final presentation: July 2nd, 2018
Description: On the options market the spreads can be high and the number of shares is quite low. There are just too little traders interested in options for it to be a low cost and effective market. Optiver aims to act as a market maker, providing liquidity to the market. It both offers to sell and buy instruments for prices that are reasonably close to each other. Optiver needs to have a good calculation of the theoretical price of an instrument, and competes with other market makers to do trades. Constantly putting in orders and monitoring them is done using algorithms, which can do this automatically and efficiently. These algorithms could potentially make mistakes, when something unforeseen happens or when there is a bug in the code. Other systems are needed to monitor the risks these algorithms are taking. This new system is a third line of defense, a post-process verification system, acting on the communication between the traders of Optiver and the market, to establish if any of these limits are broken. This system is expected to run over night to verify the over 100 million trade requests that accumulate during the morning, day, and evening trading sessions.
This project required the acquisition of explicit knowledge of the financial market, optimization techniques, advanced debugging skills, and data analysis in order to meet this goal. The project mostly acquired hinder due to some lack of knowledge on the specifics of certain markets, and the presence of data inconsistencies.
With the completion of the project, the product is expected to run daily as a check on the existing risk control systems.
Members of the project team:
Name: Eric Cornelissen
Interests: Back-end development, Software Quality, Data Science
Roles: Developer, Head Quality Control, RiskGuard Specialist
Contributions: Code Quality Control, Extensive code review, RiskGuard API development

Name: Joost Verbraeken
Interests: Software Architecture, Data Science, Finance
Roles: Developer, Lead Tester, Limit Specialist
Contributions: Testing, Optimization, Ticker Tape development, Limit checking

Name: Cornel de Vroomen
Interests: Trading, Finance, Data Science
Roles: Developer, Head Communications, AuditRecord Specialist
Contributions: Breach validation, Ledger development, organization/communication

Name: Nick Winnubst
Interests: Back-end development, Architecture Design, Finance, Multimedia Computing
Roles: Developer, Software Architect, OrderLog Specialist
Contributions: Architecture Design, Testing, Parallelization

All team members contributed to preparing the report, the final project presentation, general development of the program, and bug fixing.
Client and Coach
Name and affiliation of the client: Kris Manios and David Martens, Optiver, Amsterdam
Name and affiliation of the project coach: Georgios Gousios, EWI - TU Delft, Delft

Contact:
Eric Cornelissen, ericornelissen@gmail.com
Cornel de Vroomen, corneldevroomen@live.nl
Nick Winnubst, nwinnubst@gmail.com
Joost Verbraeken, mail@joostverbraeken.nl

The final report for this project can be found at: http://repository.tudelft.nl

Appendix B. Data Examples

B.1: Order log

```
18:44:29.208111750 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=HSIJ
K0506|KIND=Put|USER=AF0101CC002_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=88.000000|EXPIRATION=20180213|REQUESTTYPE=Inser
t|SIDE=Bid|PRICE=0.0050000000|VOLUME=10|TIMESTAMP=1518600080307002551|LIFESPAN=FAK|REQUESTID=I173D30900001809|TRACKERID=408
1|SESSIONID=7GU9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=1073780848
|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=10|PASSIVEONLY=False|SELFEXECPRE
V=True|RESTRICTION=VFCR
18:44:29.208125425 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=HSIJ
K0880|ORDERID=881870595115|TRACKERID=4081|REQUESTID=I173D30900001809|VOLUME=10|PRICE=0.0050000000|CHANGEREASON=Inserted|TIM
ESTAMP=1518600080307005224
18:44:29.208165869 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=HSIJ
K0880|ORDERID=881870595115|TRACKERID=4081|VOLUME=10|PRICE=0.0050000000|TIMESTAMP=1518600080307005224|CHANGEREASON=Expired
18:44:29.299167291 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=BHUX
K1270|KIND=Call|USER=AF0101CC001_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=127.000000|EXPIRATION=20180219|REQUESTTYPE=Ins
ert|SIDE=Bid|PRICE=0.1700000000|VOLUME=20|TIMESTAMP=1518600080398060704|LIFESPAN=FAK|REQUESTID=I073D30900002343|TRACKERID=4
082|SESSIONID=8YL9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=10737808
48|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=20|PASSIVEONLY=False|SELFEXECP
REV=True|RESTRICTION=VFCR
18:44:29.299174073 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=BHUX
K1270|ORDERID=528358603036|TRACKERID=4082|REQUESTID=I073D30900002343|VOLUME=20|PRICE=0.1700000000|CHANGEREASON=Inserted|TIM
ESTAMP=1518600080398062422
18:44:29.299198229 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=BHUX
K1270|ORDERID=528358603036|TRACKERID=4082|VOLUME=20|PRICE=0.1700000000|TIMESTAMP=1518600080398062422|CHANGEREASON=Expired
18:44:29.316081078 [Info ] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=GLS93
K1380|KIND=Put|USER=AF0101CC003_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=138.000000|EXPIRATION=20180305|REQUESTTYPE=Inse
rt|SIDE=Ask|PRICE=0.3000000000|VOLUME=10|TIMESTAMP=1518600080415974781|LIFESPAN=FAK|REQUESTID=I273D30900001837|TRACKERID=40
83|SESSIONID=8YL9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=107378084
8|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=10|PASSIVEONLY=False|SELFEXECPR
EV=True|RESTRICTION=VFCR
```

B.2: Limits

Id	Market	Underlying	Limit_role	Start_date_time	End_date_time	Order_limit	Volume_limit	Max_trades_ volume	Max_trades_ volume_window	Limit_role _id
282565	XAMS	ASML	manual	2013-07-12 14:39:57.107999	2013-07-31 17:53:05.551	6026	50400	4663	1	1
272457	XLOM	KGF	auto	2013-07-12 16:23:43.107980		7459	27341	7970	1	34
839313	XETR	SIE	manual	2013-07-12 15:25:02.103093		4900	88836	6060	1	1
838356	XLON	IMI	auto	2013-07-12 18:46:15.104097		1000	27482	5235	2	4

B.3: RiskGuard logs

```
00:05:05.731298397 [Info  ] [RISK_GUARD_SERVER] Internal market received.(Y)
00:05:05.731300891 [Info  ] [RISK_GUARD_SERVER] Receive InternalMarket: {mic="ZGDX" action=ACTION_ADD}
00:05:05.731304553 [Info  ] [RISK_GUARD_SERVER] Receive InternalMarket: {mic="ZOBI" action=ACTION_ADD}
00:05:05.731093110 [Info  ] [RISK_GUARD_SERVER] Exchange rate changes received.(Y)
00:05:05.731094748 [Info  ] [RISK_GUARD_SERVER] Receive Rate: {currency="EUR" rate=1 action=ACTION_ADD}
00:05:05.731101331 [Info  ] [RISK_GUARD_SERVER] Receive Rate: {currency="GBp" rate=87.732
action=ACTION_ADD}
12:12:12.277778320 [Info ] [RISK_GUARD_SERVER] Receive LimitUsageStatus: {market="XCME"
underlying="IPS_NOL" trading_type=TRADING_TYPE_AUTO role_id=42 username="EQ0401CC012_AQ_USTMAIN_MM"
instance_name="eml_co_cme_aucm_001_sg-5.4.4" server_id=10 status=USAGE_ACTIVE date_time=1519910996
action=ACTION_CHG}
12:12:12.277784940 [Info ] [RISK_GUARD_SERVER] CheckForNewUsage Limit usage is already added XCME-IPS_NOL-
1-87-EQ0401CC012_AQ_USTMAIN_MM-eml_co_cme_aucm_001_sg-5.4.4
12:12:12.277789702 [Info ] [RISK_GUARD_SERVER] Total Limit Usage XCME-IPS_NOL-1-87 2/4.000000
12:12:12.287406818 [Info ] [RISK_GUARD_SERVER] LimitUsageStatus received.(Y)
```

B.4: Audit records


```
{
	"message": {
		"instruments": [{
			"instrument_id": {
				"feedcode": "AEX.NL0000370179.EUR",
				"market": "AEX"
			},
			"kind": "SPOT",
			"spot_attributes": {
				"underlying": "BARK",
				"multiplier": 9001,
				"currency": "EUR"
			}
		}],
		"market": "AEX"
	},
	"message_type": "MSG_INSTRUMENTS_BY_MARKET"
}
{
	"message": {
		"instruments": [{
			"instrument_id": {
				"feedcode": "CAC.NL0000370179.EUR",
				"market": "CAC"
			},
			"kind": "BOND",
			"bond_attributes": {
				"underlying": "OOH",
				"multiplier": 3.14,
				"currency": "EUR"
			}
		}],
		"market": "CAC"
	},
	"message_type": "MSG_INSTRUMENTS_BY_MARKET"
}
{
	"message": {
		"instruments": [{
			"instrument_id": {
				"feedcode": "LSE.NL0000370179.EUR",
```

```
                        "market": "LSE"
                    },
                    "kind": "OPTION",
                    "option_attributes": {
                        "underlying": "BWAK",
                        "multiplier": 42,
                        "currency": "GBP"
                    }
                }],
                "market": "LSE"
        },
        "message_type": "MSG_INSTRUMENTS_BY_MARKET"
}
{
        "message": {
                "instrument": {
                    "combo_attributes": {
                        "underlying": "VIX",
                        "multiplier": 42,
                        "currency": "USD"
                    },
                    "instrument_id": {
                        "feedcode": "VX.S.1801.494037714",
                        "market": "XCBF"
                    },
                    "kind": "COMBO"
                }
        },
        "message_type": "MSG_INSTRUMENT_INSERTED"
}
```

Appendix C. Breaches Brazilian market

DATE	TIMESTAMP	MARKET	UNDERLYING	INSTRUMENT	ROLE_ID	TRADING_TYPE	USER	BREACHED_LIMIT	LIMIT_VALUE	REAL_VALUE
26-02-2018	14:38:08.033	XBSP	BOVA11	BOVA11PE2018031981.0000	1	auto	BQ0010PB002_AQ_BOVA11_MM	max_gross_volume_outstanding_orders	2500000	2516300
26-02-2018	14:38:08.033	XBSP	BOVA11	BOVA11PE2018031980.0000	1	auto	BQ0010PB002_AQ_BOVA11_MM	max_gross_volume_outstanding_orders	2500000	2536300
26-02-2018	14:38:08.033	XBSP	BOVA11	BOVA11CA2018031982.0000	1	auto	BQ0010PB002_AQ_BOVA11_MM	max_gross_volume_outstanding_orders	2500000	2546300
26-02-2018	20:30:38.224	XBSP	IBOV	IBOV11CE2018061389000.0000	1	auto	BQ0010PB001_AQ_IBOV_MM	max_gross_volume_outstanding_orders	20000	20020
26-02-2018	20:30:38.224	XBSP	IBOV	IBOV11CE2018061389000.0000	1	auto	BQ0010PB001_AQ_IBOV_MM	max_gross_volume_outstanding_orders	20000	20120
26-02-2018	20:30:38.224	XBSP	IBOV	IBOV11CE2018041891000.0000	1	auto	BQ0010PB001_AQ_IBOV_MM	max_gross_volume_outstanding_orders	20000	20220

Appendix D. Request without reply logs

```
11:10:42.578747587 [Info ] [ORDER_LOGGING_AUDIT]
TYPE=QuoteRequest|MARKET=ZBRQ|FEEDCODE=XLON.IE00B1FZSB30.GBP|KIND=ETF|USER=RF0001AA001_AC_WHOLESALEARB1_MM|PORTFOLIO=030264
001|SIDE=Both|BIDPRICE=13.0275|BIDVOLUME=13550|BIDTRACKERID=478|ASKPRICE=13.04|ASKVOLUME=13550|ASKTRACKERID=479|REQUESTID=R
FQ_ETF_OPTV_875776_1|TIMESTAMP=1519899042578724000|SESSIONID=OPTVETFRFQ→BLPRFQ|TRADERKEY=danielkrycha|MLALGOID=2181048320|D
ECISIONID=2181048320|PRICETYPE=MONE|PRICECCY=GBP|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=Rfq|PASSIVEONLY=False|SELFEX
ECPREV=True|RESTRICTION=VFCR
11:10:43.000286272 [Info ] [ORDER_LOGGING_AUDIT]
TYPE=QuoteUpdateReply|MARKET=ZBRQ|FEEDCODE=XLON.IE00B1FZSB30.GBP|SIDE=Bid|BIDPRICE=13.0275|BIDVOLUME=13550|BIDTRACKERID=478
|BIDQUOTEID=RFQ_ETF_OPTV_875776|BIDCHANGEREASON=Inserted|REQUESTID=RFQ_ETF_OPTV_875776_1|TIMESTAMP=1519899043000267000
11:10:43.000298701 [Info ] [ORDER_LOGGING_AUDIT]
TYPE=QuoteUpdateReply|MARKET=ZBRQ|FEEDCODE=XLON.IE00B1FZSB30.GBP|SIDE=Ask|ASKPRICE=13.04|ASKVOLUME=13550|ASKTRACKERID=479|A
SKQUOTEID=RFQ_ETF_OPTV_875776|ASKCHANGEREASON=Inserted|REQUESTID=RFQ_ETF_OPTV_875776_1|TIMESTAMP=1519899043000289000
16:08:03.154876464 [Info ] [ORDER_LOGGING_AUDIT]
TYPE=QuoteRequest|MARKET=ZBRQ|FEEDCODE=XLON.IE00B1FZSB30.GBP|KIND=ETF|USER=RF0001AA001_AC_WHOLESALEARB1_MM|PORTFOLIO=030264
001|SIDE=Ask|ASKPRICE=13.03|ASKVOLUME=11350|ASKTRACKERID=277|REQUESTID=RFQ_ETF_OPTV_877209_1|TIMESTAMP=1519916883154854000|
SESSIONID=OPTVETFRFQ--
>BLPRFQ|TRADERKEY=danielkrycha|MLALGOID=2181048320|DECISIONID=2181048320|PRICETYPE=MONE|PRICECCY=GBP|VOLUMETYPE=UNIT|LIQUID
ITYPROV=False|ORDERTYPE=Rfq|PASSIVEONLY=False|SELFEXECPREV=True|RESTRICTION=VFCR
```

Appendix E. Research Report

Introduction

Optiver is a liquidity provider on the global stock markets. They trade the markets with their own money and are always willing to buy products from you or sell products to you. Trading all these millions of products is impossible to do by hand, instead Optiver uses sophisticated algorithms which automatically keep track of the vast amount of trades and products. To be sure that these algorithms do not lose control certain limits are imposed. For example, these limits include the maximum amount of orders to be send out per second. To make sure these limits are adhered to, we will create a program to analyze historical data and check if any breaches have occurred, initially on data from the 2nd of January until the 16th of April. The biggest challenge will be to go through the data in a reasonable time frame. Our time frame, which consists of about 80 days, has in total about 16 terabytes of data, which is about 200 GB of data per day. Another challenge is that the limits that should be checked are context-sensitive, which means that we have to do a simulation on the outstanding orders to know what the conditions are at a certain time.

Coach and Client

Our coach is Dr. Georgios Gousios from the TU Delft and the client is Optiver, represented by Kris Manios and David Martens.
Georgios Gousios specializes in Software Analytics, which help software developers to get statistics about their programs and act on these to create better quality software. We've asked Georgios to be our coach because of his knowledge about software engineering and data science, two important relevant fields for this project.
Optiver is a leading global electronic market maker, focused on pricing, execution and risk management. Optiver provides liquidity to financial markets using its own capital, at its own risk, trading a wide range of products: from listed derivatives, cash equities, ETFs, and bonds to foreign exchange. Optivers independence allows them to objectively improve the markets and provide efficiencies for market participants.

Project

Description & value proposition

Optiver has a lot of algorithms running which automatically or semi-automatically trade the global market. The tech department together with the risk department have included all kinds of limits in these algorithms. This can be limits to manage risk of the portfolio or limits imposed by an exchange. An example of these limits is a limit on the maximum outstanding volume for a certain instrument. For example, an option on ASML has the limit of 10,000 lots outstanding volume, which means that the total amount of lots on an exchange of ASML should not be larger than 10,000. Some of these limits can become very complicated, and it can be difficult to verify that they work correctly just by looking at the code. Our client Optiver wants to know if these limits were ever exceeded, and potentially make a program which would check for these breaches on a daily basis.
The net value of the project is found in this presence or absence of these breaches. This can help the risk department identify the sources of potentially dangerous exposure to the market. Furthermore, the project serves as a check on the current limit systems used on the trading floor.
The project involves the data analysis of the orders send to exchanges this year, such that we can see if any of the limits were broken in the past. The data sets that is used to do this are the order and trade logs generated for compliance purposes. The challenge here lies in the size of these logs, and that the logs are not designed to do research on. The size of the logs amounts to about 200 GB per day, and to analyze this in a reasonable time frame we need to find ways to speed up the process, for example by distributed computing. The other challenge is that the data is not designed to do analysis on, and we could potentially get stuck on the verification of certain limits or the accuracy of the verification, when for example certain data or mappings are missing.
If the project is finished in less than 10 weeks time, we can extend the project by providing daily, or even intraday, insight into the broken limits. That would make it easier to spot and fix new breaches.

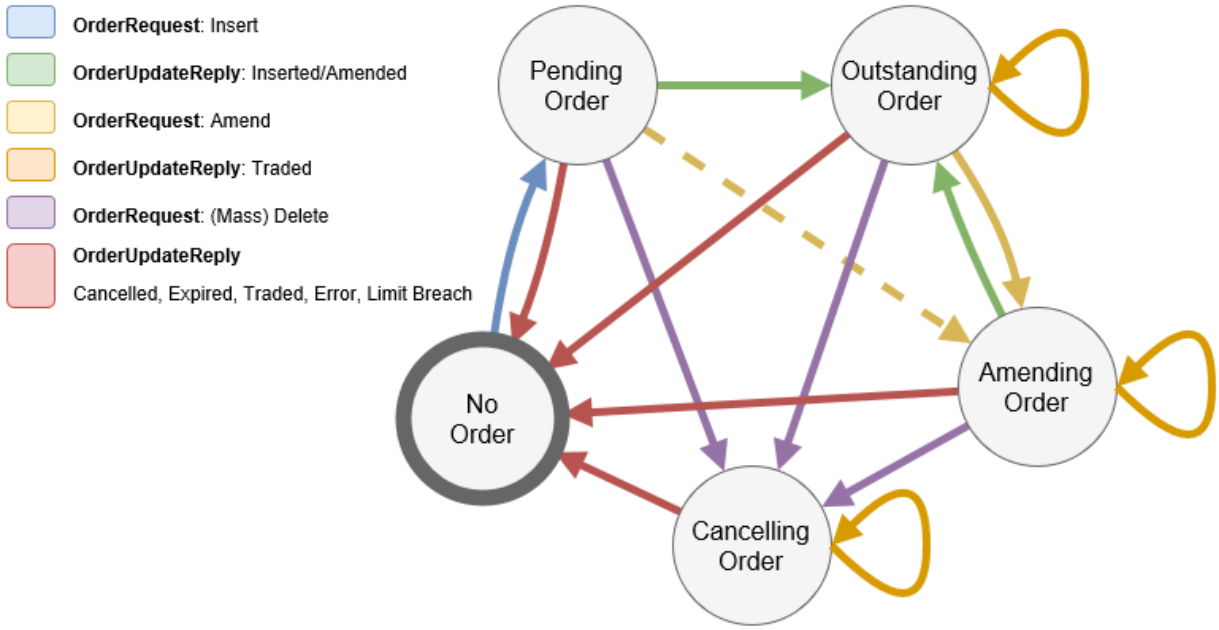
Order log books

Description

The order log books consist of all the orders sent to exchanges and their respective responses. Every time an order is sent to an exchange, the attributes of that order are logged. When the response of the exchange is received, for example that an order is placed in the order book, another line is created. Finally messages from the exchange like an expired or cancelled order are also written down in this log.
Below is a snippet of a log (data is randomized):

18:44:29.208111750 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=HSIJ K0506|KIND=Put|USER=AF0101CC002_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=88.000000|EXPIRATION=20180213|REQUESTTYPE=Insert|SIDE=Bid|PRICE=0.0050000000|VOLUME=10|TIMESTAMP=1518600080307002551|LIFESPAN=FAK|REQUESTID=I173D30900001809|TRACKERID=4081|SESSIONID=7GU9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=1073780848|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=10|PASSIVEONLY=False|SELFEXECPREV=True|RESTRICTION=VFCR
18:44:29.208125425 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=HSIJ K0880|ORDERID=881870595115|TRACKERID=4081|REQUESTID=I173D30900001809|VOLUME=10|PRICE=0.0050000000|CHANGEREASON=Inserted|TIMESTAMP=1518600080307005224
18:44:29.208165869 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=HSIJ K0880|ORDERID=881870595115|TRACKERID=4081|VOLUME=10|PRICE=0.0050000000|TIMESTAMP=1518600080307005224|CHANGEREASON=Expired
18:44:29.299167291 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=BHUX K1270|KIND=Call|USER=AF0101CC001_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=127.000000|EXPIRATION=20180219|REQUESTTYPE=Insert|SIDE=Bid|PRICE=0.1700000000|VOLUME=20|TIMESTAMP=1518600080398060704|LIFESPAN=FAK|REQUESTID=I073D30900002343|TRACKERID=4082|SESSIONID=8YL9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=1073780848|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=20|PASSIVEONLY=False|SELFEXECPREV=True|RESTRICTION=VFCR
18:44:29.299174073 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=BHUX K1270|ORDERID=528358603036|TRACKERID=4082|REQUESTID=I073D30900002343|VOLUME=20|PRICE=0.1700000000|CHANGEREASON=Inserted|TIMESTAMP=1518600080398062422
18:44:29.299198229 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderUpdateReply|MARKET=XCME|FEEDCODE=BHUX K1270|ORDERID=528358603036|TRACKERID=4082|VOLUME=20|PRICE=0.1700000000|TIMESTAMP=1518800080398062422|CHANGEREASON=Expired
18:44:29.316081078 [Info] [ORDER_LOGGING_AUDIT] [fex] TYPE=OrderRequest|MARKET=XCME|FEEDCODE=GLS93 K1380|KIND=Put|USER=AF0101CC003_AT_FXMAIN_MM|PORTFOLIO=04003226|STRIKEPRICE=138.000000|EXPIRATION=20180305|REQUESTTYPE=Insert|SIDE=Ask|PRICE=0.3000000000|VOLUME=10|TIMESTAMP=1518600080415974781|LIFESPAN=FAK|REQUESTID=I273D30900001837|TRACKERID=4083|SESSIONID=8YL9X0N|TRADERKEY=MP_TT_AT_1|MLALGOID=1073780848|ACCOUNTTYPE=CTI2|CUSTOMERORFIRM=Customer|DECISIONID=1073780848|PRICETYPE=MONE|PRICECCY=USD|VOLUMETYPE=UNIT|LIQUIDITYPROV=False|ORDERTYPE=2|DISPLAYVOLUME=10|PASSIVEONLY=False|SELFEXECPREV=True|RESTRICTION=VFCR

The log is in a key-value pipe delimited format. Every line either involves a request to an exchange or a reply from an exchange. Because the type of the product can differ per line, they do not necessarily have the same attributes. A simulation of these orders over time has to be made such that we know what order conditions (e.g. outstanding volume or amount of active orders) there are at any moment. For checking the limits we need the attributes like VOLUME and PRICE. The attributes like MARKET, FEEDCODE and TRACKERID will be used to map it to the limits and to the product specification. Finally, the time at the start of the line is critical in order to check which limit was active at that moment and to verify time-based limits. The feedcode is an internal code used to identify products, and the underlying product does not become clear from the loglines. As the underlying is what all the limits are for, we need a mapping from feedcode to underlying, which is what we will use the audit records for.
Orders can be in multiple states, the full life cycle of an order is illustrated in the following diagram:



Files

The trade logfiles are gzipped and structured in directories based on year, month and day number. The size of these compressed files ranges from several kilobytes to 300 megabytes, with a compression ratio of 7%. This results in files of up to 4GB uncompressed. This is the maximum size of a log file, if that limit is reached the log continues in a new file. Although this limit is rarely reached, it should be considered when analyzing the files. The file name (e.g. "20180302-1830-20180302-0830-afex_co_at_aucm_201.OrderLoggingAudit") contains the start and end time of the log, for example when a log is from 08:30 to 18:30, those times are included in the name. It also contains the code for a certain market link and the instantiation number. Note that market links can have multiple instantiations which can lead to multiple messages being sent at the same time. These files can contain 7 types of entries, namely: *OrderLoggingInfo, OrderRequest, OrderUpdateReplay*, *QuoteRequest*, *QuoteUpdateReply*, *MassDelete*, and *Position*.

- OrderLoggingInfo indicate the start of a logging file.
- OrderRequest are send by Optiver and contain details about the order placed, such as the market, volume, price, and lifespan.
- OrderUpdateReply come from the market and inform about what happened to a particular order (identified by the request id), for example if the order is inserted, expired, or sold.
- QuoteRequest, similar to order logs, are send by Optiver with information about the quote which we will use to check the limits.
- QuoteUpdateReply, similar to order updates, come from the market with updates about a particular order (again identified by the request id).
- MassDelete entries are requests to quickly delete a group of quotes.
- Position log entries can be ignored for the purposes of this project.

Limits

Description

To check if any breaches have occurred, we need to know which limits were active and at which moment. These limits are regularly changed, and an overview exists in which every historical limit over a certain time window is kept together with the start and end date. This file is CSV formatted like the sample below (data is randomized):

id, market, underlying, limit_role, change_user, change_comment, order_limit, volume_limit, orders_per_second, multiple_exchanges, derivative, max_trades_per_second, max_trades_per_day
282565, XAMS, ASML, manual, name, comment, 6026, 50400, 250, 1, 1, 4663, 25854

272457, XLON, KGF, auto, name, comment, 58279, 7459, 27341, 0, 0, 7970, 4920
839313, XETR, SIE, manual, name, comment, 2525900, 4900, 88836, 1, 0, 6060, 745
838356, XLON, IMI, auto, name, comment, 58295, 1000, 27482, 1, 1, 5235, 663

Every entry in this CSV can be identified either by the id, or by a combination of the market, underlying, and limit_role. Most of the in total 35 columns are relevant for the purpose of our project, but some can be ignored. The full limits history is about 300,000 rows, which gives a rough indication of the size of this data, we can easily read it and then keep its contents in memory during the program.

Files

All this data can be found in a single CSV file. The so-called *limits history* file contains a list of all limits sorted on time. We will identify a certain limit by using the combination of market, underlying product, limit role, limit role id and time which it is active.

Risk Guard

Description

Risk Guard is an internal system at Optiver which (among other things) accumulates the active limits on start up and can then be called by other systems to request access to a certain limit. Before any trades are made, the market link first retrieves the limits from the risk guard, which will then be used to prevent . These are raw log files, and contain information like in the following snippet (data is randomized):

12:12:12.277778320 [Info] [RISK_GUARD_SERVER] Receive LimitUsageStatus: {market="XCME" underlying="IPS_NOL" trading_type=TRADING_TYPE_AUTO role_id=87 username="EQ0401CC012_AQ_USTMAIN_MM" instance_name="eml_co_cme_aucm_001_sg-5.4.4" server_id=10 status=USAGE_ACTIVE date_time=1519910996 action=ACTION_CHG}
12:12:12.277784940 [Info] [RISK_GUARD_SERVER] CheckForNewUsage Limit usage is already added XCME-IPS_NOL-1-87-EQ0401CC012_AQ_USTMAIN_MM-eml_co_cme_aucm_001_sg-5.4.4
12:12:12.277789702 [Info] [RISK_GUARD_SERVER] Total Limit Usage XCME-IPS_NOL-1-87 2/4.000000
12:12:12.287406818 [Info] [RISK_GUARD_SERVER] LimitUsageStatus received.(Y)

Each log entry can be one of many types, explained in more detail below, and the information in it can be used to determine what limit was used by a particular auto-trader at a particular point in time. The last change logged in the file is the limit imposed at that moment for an underlying-market pair.

The log files from the Risk Guard contain all the information described above, plus some extra data. For the purposes of our project we're only interested in loglines containing the RISK_GUARD_SERVER identifier and can ignore the other lines. The Risk Guard will log different kinds of things including the limits it received, requests for limits and breach reports. The relevant log entries for our project are listed here, other log entries can be ignored:

- *Receive Limit*: Receive limits from the limit database, contains the data found in the limit files.
- *Receive LimitUsageStatus*: Call to get limit usage status. Contains the market, underlying, trading_type, role_id, username, instance_name, server_id, status, date_time, and action. Followed by either CheckForNewUsage or an EraseUsage log entry.
- *Receive LimitUsageRequest*: Call to receive limit usage. Contains the market, underlying, trading_type, role_id, username, and instance_name. Followed by
    - Receive Limit usage | Send LimitUsageStatus | LimitUsageStatus received
- Receive Breach: breached detected by the market link. Contains the reach_id, market, underlying, trading_type, username, component, brach_date_time, breach_value, limit_name, limit_value, role, feedcode, action. Followed by either:
    - *nothing*
    - Get Breaches Report | Send breaches

Files

For the Risk Guard logs we have the full daily log like the example data given. As there are two Risk Guard machines running, we have two files which should either be put together before analysis, or the output of the parsing should be merged.

Others

Besides that, there are a couple of other files which are needed to fully analyze the limits against the order logs:

- Audit records: A list of changes to the mapping from feedcodes to underlying and market. This data set consists of a snapshot with all instruments, and daily additions to this set. We will use this to retrieve the underlying and the market when we have the feedcode.
- Internal markets table: Besides the log files, the Risk Guard system also provides an overview of the internal market used by Optiver. There is no need for us to analyze the internal markets, so with this information we can determine if an order or quote can be ignored, hence speeding up our processing time.
- Limit roles table: A limit can have multiple roles in order to grant the ability to allow specific traders more leeway. This means that a single market-underlying combination can have multiple limits at the same time, depending on whom is trading. Details about these roles can be found in this table.

Overlap

The different data files can be linked to each other through shared identifiers. The table below shows relevant data points for us that we can use to combine data in order to analyze it.

	Explanation	**Order logging - Request**	**Order logging - Reply**	**Risk Guard**	**Limit history**	**Limit**	**AuditRecords**	**Roles table**	**Internal markets table**
Timestamp	Limits change over time, so this is required to determine the limit	X	X	X	X				
Market	Orders are to be analyzed per market-underlying combination	X	X	X	X	X	X		X
Underlying	Orders are to be analyzed per market-underlying combination			X	X	X	X		
Feedcode	A code representing the market and underlying	X	X				X		

Limit trading type	The type of trader for which this limit applies, i.e. "manual", "auto", or "beta"			X	X	X			
Limit trading role id	Limits for an order are tied to a role			X (role_id)	X	X		X	
Username	The name of the algorithm	X (user)		X					
Instance name	The instance name of the algorithm	X (in filename)		X					
Order id	Identifier of a particular order	X	X						

Flaws

The log files and databases provided by Optiver are not designed for the purpose of analyzing whether or not limits are exceeded by the auto-traders. Like a cartographer, we need to explore the surrounding landscape in order to make a map. Similarly, we need to combine the existing partial maps to create a sound whole. In essence, this in and of its self is the problem that we are tackling. We need to connect a number of loose dots between the different files in order to properly analyze the data. This requires us to do a lot of prospecting before even knowing what our landscape looks like. An example of this is the fact that the limit applicable to a certain order cannot be derived from just the order log and limit history, instead we have to analyze the Risk Guard log files to link the order to the limit with the correct limit role.

Implementation

Prototype

To first see how the data works, and to try testing some of the limits on small amounts of data, we will use Python to parse and go through the data. This is chosen for two reasons, the first one being that Python is a language which is fast to develop such a prototype in. For a prototype, it is important to iterate quickly to see which design choices are useful and which are not useful to include in the final design. The second reason to use Python is the available expertise within Optiver, where the projects are developed exclusively in C++ and Python, with some C# for visualization. The other reason is that Optiver has a pre-existing parser and validation system for the order logs.
We can give an argument against the use of Python, because speed is important when analyzing massive amounts of data and Python is a lot slower (up to a few hundred times) than other programming languages such as C++ [1]. However, for a prototype the speed is not really important, and in the real program the heavy data processing will be done with the Apache Flink framework which is implemented in Java, and is a lot faster than Python [2].

Architecture

To design the program we created a flowchart (divided in pre-processing, limit analysis and results) and an overview of the program architecture based on the flowchart, they can be found in Figure 1 and Figure 2.

The top part of the program architecture diagram is the access point of the program, the left half parses the limits and feedcodes and the right half are the limit verifiers, which run in parallel per day and per market instance. The program will be developed in a bottom-up approach where we start by processing the data and then verify the data in chronological order. The 4 data files are the (historical) risk limits, the RiskGuard logs, the Market Instance logs, and the audit records. After parsing the files and extracting the information we need, we split it up into two sets. Firstly, we derive today's trading limits per auto-trader role. This is derived by combining data from the historical limits and the RiskGuard logs. Then we map the feedcodes for the given day to their underlying product. Finally, we derive the set of currently outstanding orders in order to check the in-trade limits, and a list of recent trades in order to verify the pre- and post-trade limits. Due to the fact that every Market Instance is individually bounded by a set of limits, and that these limits are 'reset' at the end of the day, this processing can be parallelized per day and per Market Instance. This should be facilitated by a pool management system that is in direct control of the main program. After the analysis has finished, the breaches are stored in a log file for further evaluation.

Framework

Since the final goal is to process the logs per day, and potentially as it is generated as well, we want to use a processing framework that suits this need. A common approach for analyzing large volumes of data is MapReduce[3], popularized by Google in the early 2000's. Some good and well known frameworks for MapReduce analytics are freely available such as Apache Hadoop[4][5] and Disco MapReduce[6][7]. However, MapReduce is a so-called *batch-processing* approach, meaning that data is analyzed in a large volume at once, which is well suited for per day analysis of the data. The alternative to batch-processing is *stream-processing*, where data is analyzed as it is being generated. As a newer concept, stream-processing does not have a standard approach but a number of frameworks do exist that allow you to perform distributed stream-processing, such as Apache Kafka[8][9], Apache Storm[10][11], Apache Spark[12][13], and Apache Flink[14][15]. For the purpose of this research project we want a system that is capable of both batch- and stream-processing.

Apache Storm is designed to process real-time data in the form of tuples. The maintainers claim reliable processing of real-time unbounded streams of data, clocked in a benchmark at over a million tuples processed per second per node. Iqbal and Soomra [15] indicate that setting up Storm can be somewhat difficult. Storm is compatible with a multitude of languages, among which are Java and Python. Unfortunately, it does not have proper native support for batch processing, so performing an analysis of data from a previous day (in batch) is not very efficient. Although the log data we have to analyze would fit the tuples approach of Storm, hence we decided not to use it because other systems provide both stream and batch processing support.

The distributed streaming platform Apache Kafka integrates stream processors with the Publish-Subscribe pattern[17] and is designed partly with log processing in mind. It is capable of stream processing over a hundred gigabytes of data a day but does not provide native support for batch processing. It is a widely used platform[18][19][20][21] designed to be used with Java or Scala. Kafka shows[22] good results in benchmarking tests with regards to the end-to-end latency of its consumers, producers and streams. We decided not to use Apache Kafka because the Publish-Subscribe pattern does not fit our project and it also lacks proper native support for batch processing.

The on Resilient Distributed Datasets (RDDs) build cluster-computing system Apache Spark is designed for both batch and streaming data. Its main components are a DAG scheduler, a query optimizer, and a physical execution engine and it can be used interactively with a number of languages including Python and Scala, even allowing a combination of SQL querying and stream processing. The creators of Spark claim that it outperforms Hadoop in a certain analytics problems and can be used to interactively query a 39 GB dataset within a second. Resilient Distributed Datasets are a (fault-tolerant) distributed memory abstraction, allowing large cluster to perform computations on in-memory data[23].

Apache Flink is similar to Apache Spark in that both provide an API for both batch jobs and stream jobs. Flink interfaces with Scala and Python and supports a multitude of data processing approaches. It is being used by, among others, *Alibaba.com* and *Uber* for use-cases such as optimizing e-commerce search

results and data science. Apache Flink reportedly[24][25] has lower latency then Apache Spark, as well as better performance when batch processing (certain data). On top of that, Flink provides a richer API than Spark. Hence we decided to use Apache Flink for processing and analysing the datalogs both in real-time and when batch processing is preferred.
Furthermore, there exist resource management systems such as Apache Mesos[26][27] which can be used in conjuction with the abovementioned processing platforms. We decided not to use such a system as it is beyond the scope of this project and we believe the framework by itself will prove sufficient.

Pre-existing work

Due to the data specific analysis on specific and proprietary data, no external tools exists to handle the problem. However, due to the continuous development of tools internally, there exists a foundation from a research perspective. However, the logs we use are mostly made for compliance reasons, instead of for analytical purposes, and not a lot of projects use the logs, hence the relevant tools to solve our problem are limited. Currently, the only applicable tools are protocol definitions and a module that validates order logs.

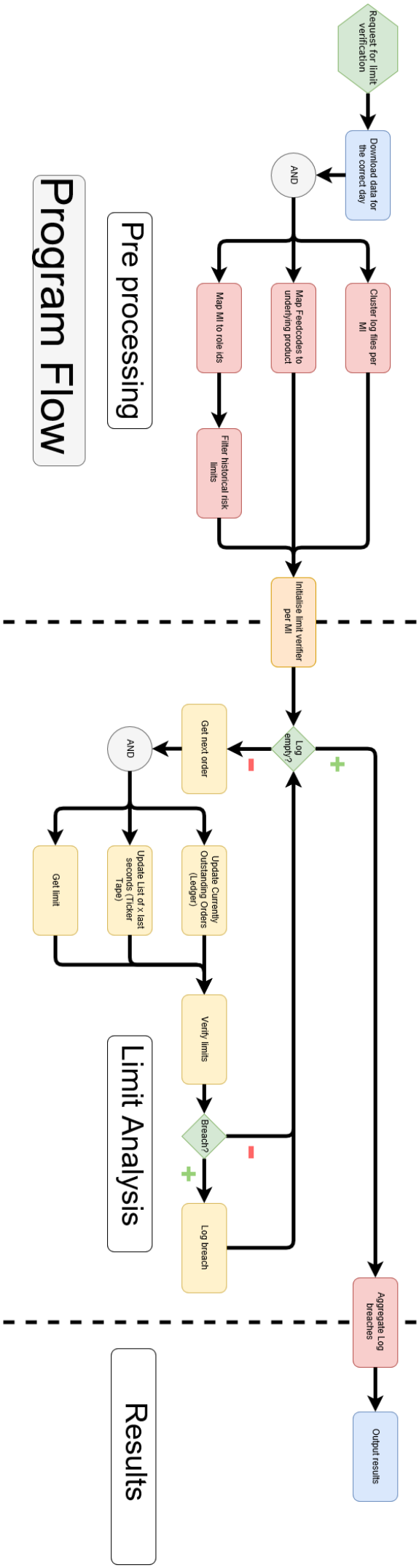
Language

The language we will write our program in will be Python. We've decided to write the both a prototype and the final implementation in Python and to use the framework Flink for our implementation.
Because we settled on using Flink for the cluster processing of the data (see the section above), we evaluated all 3 languages for which there is an API available of Flink. To choose what would be the best language to use we compared the languages based on the software development speed, the runtime performance and the experience / usage within Optiver.
The speed of software development depends on the level of the language [28] and the experience of the programmers with that language. A high-level language will have a higher development speed than a lower level language. In terms of software development speed, Java would be the slowest. It is the lowest level language of the three and requires to write a lot of boiler plate code to achieve the same result as with Python of Scala. The development speed of Python and Scala is somewhat similar.
The runtime performance (execution speed) is a crucial aspect of choosing a programming language for this project, as a large amount of data has to be processed. Scala is build upon Java, so we could say that these are somewhat similar. The comparison between Java and Python [26] shows that Java is on average 25 times faster in execution than Python. But as we are only using the API of Flink, it will probably not matter that much. Most of the work will be done by the Flink cluster (which is written in Java), and only small requests or checks will be made from our implementation.
Lastly we consider the languages used in production at Optiver. Internally, Optiver nearly exclusively develops in C++ and Python, with some C# for visualisation. As Flink does not have a C++ and C# API implementation, we do not consider those.
In conclusion, we decided to use Python, as we think it is useful that Optiver already has a lot of programs in Python and that the execution speed does not really matter when using Flink like we do. The last reason is that it can be useful to use parts from our prototyping and use that in the final program.

Front-end

The output which is generated from our data analysis has to be visualized such that breaches can be easily identified. For the main project, the batch processing of the historical data, we are going to generate a report in which it becomes clear which breaches (if any) have occurred. This should have all the necessary data such that the breaches can be found manually as well.
If we get to the part such that a daily run over the order data, we are going to show the data on a simple dashboard. As our project is mainly about analyzing the data such that we can find any breaches of risk limits, we will first focus on achieving that before we will focus on the front-end part We could even decide to not make a front-end as the logs will only be seen by other developers with a clear understanding of the limits. When we do finish the analysis and decide a dashboard is needed, a webserver in Python, for example Flask [29], could be used to show the data on.

Figure 1


Program Flow
Pre processing
Limit Analysis
Results
Request for limit verification
Download data for the correct day
AND
Map MI to role ids
Map Feedcodes to underlying product
Cluster log files per MI
Filter historical risk limits
Initialise limit verifier per MI
Log empty?
Get next order
AND
Get limit
Update List of x last seconds (Ticker Tape)
Update Currently Outstanding Orders (Ledger)
Verify limits
Breach?
Log breach
Aggregate Log breaches
Output results

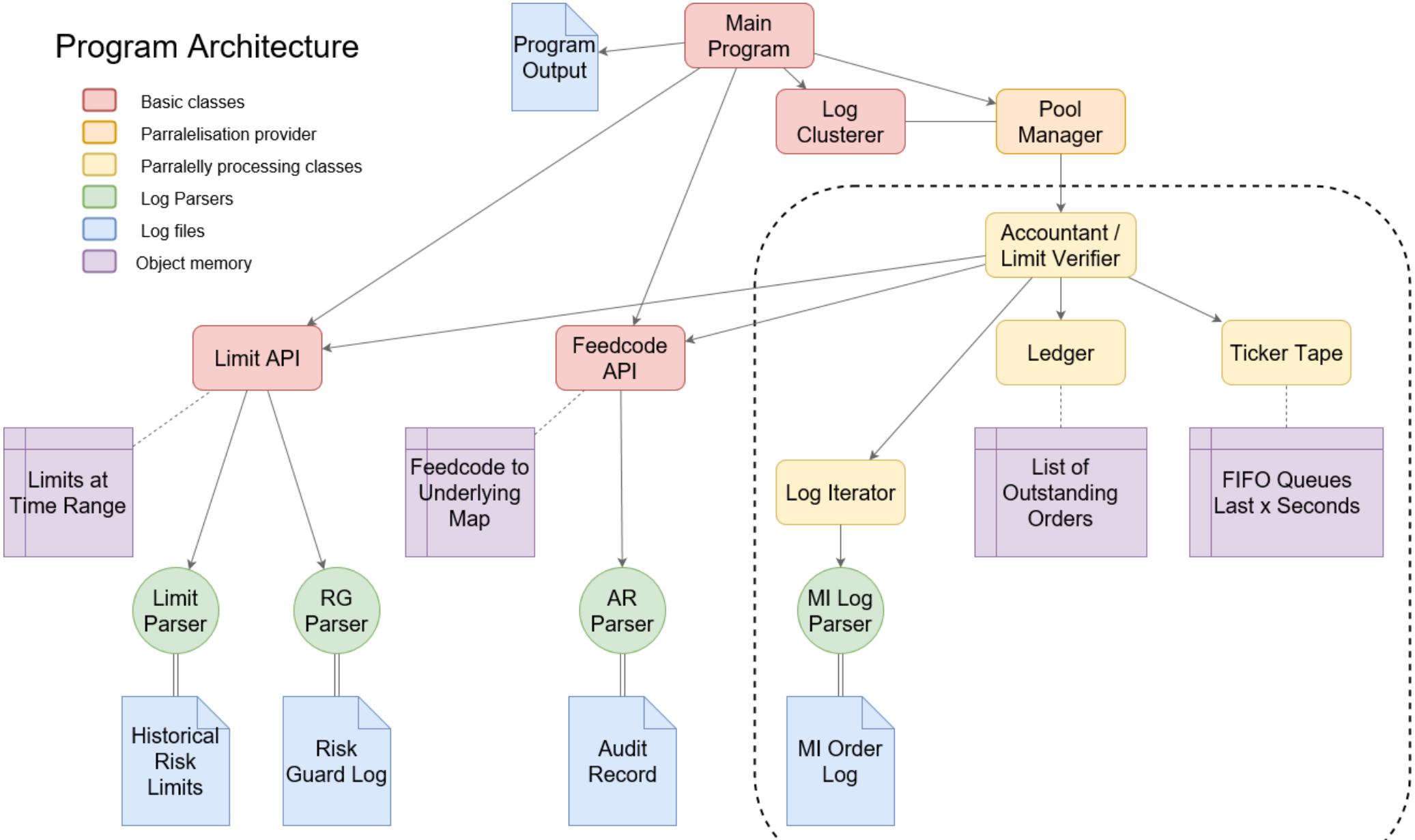

Program Architecture
Basic classes
Parralelisation provider
Parralelly processing classes
Log Parsers
Log files
Object memory
Program Output
Main Program
Log Clusterer
Pool Manager
Accountant / Limit Verifier
Limit API
Feedcode API
Ledger
Ticker Tape
Limits at Time Range
Feedcode to Underlying Map
Log Iterator
List of Outstanding Orders
FIFO Queues Last x Seconds
Limit Parser
RG Parser
AR Parser
MI Log Parser
Historical Risk Limits
Risk Guard Log
Audit Record
MI Order Log

Figure 2

References

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/python3-gpp.html
[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/python.html
[3] http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf
[4] http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-hadoop-oracle-194542.pdf
[5] http://hadoop.apache.org/
[6] http://hpds.ee.kuas.edu.tw/download/distributed_system/9802/9802final_ppt/KU%20NAI%20HUI/DisCo-Distributed%20Co-clustering%20with%20Map-Reduce.pdf
[7] http://discoproject.org/
[8] https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf
[9] https://kafka.apache.org/
[10] https://www.researchgate.net/profile/Tariq_Soomro/publication/271196175_Big_Data_Analysis_Apache_Storm_Perspective/links/54bff9900cf28a6324a02e6b.pdf
[11] http://storm.apache.org/index.html
[12] https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf
[13] http://spark.apache.org/
[14] http://www.diva-portal.org/smash/get/diva2:1059537/FULLTEXT01.pdf
[15] https://flink.apache.org/
[16] https://www.researchgate.net/profile/Tariq_Soomro/publication/271196175_Big_Data_Analysis_Apache_Storm_Perspective/links/54bff9900cf28a6324a02e6b.pdf
[17] https://infoscience.epfl.ch/record/165428/files/10.1.1.10.1076.pdf
[18] http://sites.computer.org/debull/A12june/pipeline.pdf
[19] http://www.vldb.org/pvldb/vol8/p1654-wang.pdf
[20] http://cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf
[21] https://kafka.apache.org/powered-by
[22] https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
[23] https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
[24] http://www.diva-portal.org/smash/get/diva2:1059537/FULLTEXT01.pdf
[25] https://cds.cern.ch/record/2208322/files/report.pdf
[26] https://books.google.nl/books?hl=en&lr=&id=fPwKCgAAQBAJ&oi=fnd&pg=PP1&dq=apache+mesos&ots=lcZ4F8avD9&sig=Bm_5MTuzXBQsxBSl3_osL0o4p1s#v=onepage&q=apache%20mesos&f=false
[27] http://mesos.apache.org/
[28] J. K. Ousterhout, Scripting: Higher Level Programming for the 21 Century, Computer (IEEE), 1998, https://web.stanford.edu/~ouster/cgi-bin/papers/scripting.pdf
[29] http://flask.pocoo.org/

Appendix F. Order data overview

	Explanation	OrderRequest	OrderUpdateReply	QuoteRequest	QuoteUpdate Reply
Type	Type of the order, e.g. "order request" or "update reply"	X	X	X	X
Market	Market on which the order is traded	X	X	X	X
Feedcode	A code representing the market and underlying	X	X	X	X
Kind	Bond BondFuture Call Combo DividendFuture ETF Future Fx FxForward LEPOCall Put Spot VolatilityCombo	X (if RequestType = Insert)		X	
User	The user who initiates the request	X (if RequestType is Insert)	Optional	X	Optional
Request type	• Insert • Amend • Delete	X			
Side	• Ask • Bid • Both	X (Both is not possible)	X (Both is not possible)	X	X
Price	Price of the order	X (if RequestType is Insert)	X		
Volume	Volume of the order	X (if RequestType is Insert)	X		
Timestamp	Timestamp of the order in nanoseconds; not	X	X	X	X

	synchronized across machines				
Lifespan	• **FAK**: fill and kill • **GTC**: good till cancel • **GFD**: good for day • **GTD**: good till date	X (if RequestType is Insert)	Optional		
Request ID	Sent to the exchange to match request with reply and updates	X	X	X	X (if change reason is not cancelled, expired or traded)
Tracker ID	Used to track the order during its lifecycle	X	X		
PriceCCY	Currency being used, not used by our program as we get the currency from the audit records log	X (if RequestType is Insert)		X	
Order type	Set to "Rfq" when the order is an Rfq order	X (if RequestType is Insert)		X	
Change reason	• **Unknown** - when no other value applies • **Inserted** - used when a new order was inserted • **Amended** - used when an existing order was modified • **Cancelled** - used when an existing order was cancelled • **Traded** - used when an order was amended or cancelled due to a trade • **Expired** - used when the order expired without being traded • **LimitsBreached** - used when the request		X		

	could not be completed because the predefined limits were breached • **Recovered** - when the ML restarts and then recovers a previous state of an order				
Ask price	Price at the ask side			X (if Side is Ask or Both)	X (if Side is Ask or Both)
Ask volume	For quote requests and most replies, the volume at the ask side For traded quotes, the volume that's left after the trade For traded RFQs, the volume that is traded			X (if Side is Ask or Both)	X (if Side is Ask or Both)
Bid price	Price at the bid side			X (if Side is Bid or Both)	X (if Side is Bid or Both)
Bid volume	For quote requests and most replies, the volume at the ask side For traded quotes, the volume that's left after the trade For traded RFQs, the volume that is traded			X (if Side is Bid or Both)	X (if Side is Bid or Both)
Ask quote ID	Used to match trades to quotes			X (if Side is Ask or Both)	X (if Side is Ask or Both)
Bid Quote ID	Used to match trades to quotes			X (if Side is Bid or Both)	X (if Side is Bid or Both)

Ask Change Reason	• **Unknown** - when no other value applies • **Inserted** - used when a new quote was inserted • **Amended** - used when an existing quote was modified • **Cancelled** - used when an existing quote was cancelled • **Expired** - used when the quote expired without being traded • **Traded** - used when a quote was amended or cancelled due to a trade • **LimitsBreached** - used when the request could not be completed because the predefined limits were breached				X (if Side is Ask or Both)
Bid Change Reason	• **Unknown** - when no other value applies • **Inserted** - used when a new quote was inserted • **Amended** - used when an existing quote was modified • **Cancelled** - used when an existing quote was cancelled • **Expired** - used when the quote expired without being traded • **Traded** - used when a quote was amended				X (if Side is Bid or Both)

	or cancelled due to a trade • **LimitsBreached** - used when the request could not be completed because the predefined limits were breached				

Appendix G. SIG correspondence

Feedback week 6 - Dutch

De code van het systeem scoort 3.4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Interfacing en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten.
Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden.
In jullie geval heeft de constructor van FieldDefinition in rules.py wel erg veel parameters. Je zou die configuratie eigenlijk naar parameter objecten willen uitsplitsen, om te voorkomen dat aanroepen van de constructor op termijn steeds moeilijker leesbaar worden.
Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.
Bij jullie project is verify() in accountant.py een goed voorbeeld. Dat proces bestaat uit meerdere stappen die nu met commentaar worden aangegeven. Je zou dat eigenlijk willen splitsen zodat de stappen ook apart testbaar worden.
Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.
Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

Feedback week 6 - English (translated)

The code of the system scores 3.4 stars on our maintainability model, this means that the code is maintainable on a market average level. We suggest Unit Interfacing and Unit Complexity as points of improvement given their lower partial scores.
For Unit Interfacing we look at the percentage of code with an above average number of parameters. Generally this indicates a lack of abstraction. Added to that is the fact that a large number of parameters can be confusing when calling the method and it often means methods become bigger and more complex. In your code the constructor of FieldDefinitions in rules.py has a lot of parameters. We suggest moving the configuration to parameter object to avoid that constructing an object becomes more and more difficult with time.

For Unit Complexity we look at the percentage of code that has an above average level of complexity. Splitting up such complex methods into smaller chunks ensures that every chunk is easier to understand, easier to test and as a result easier to maintain. By splitting the functionalities into separate methods with descriptive names, every part can be tested separately and the overall flow of the program will become easier to understand.
For your project the verify() method in accountant.py is a good example of this. That process consist of multiple steps that are currently denoted using comments. Preferably this is split into separate steps that are can be tested separately.
Lastly we want to say that we did not find any (unit)test-code in the upload. We strongly recommend to at least have tests defined for the important functionality to ensure that no future changes accidentally lead to unexpected behaviour.
In general there is still some improvements possible, hopefully you are able to carry out these improvements in the remaining development phase.

Appendix H. Test coverage

Total	1921	108	0	94%
Module	**Statements**	**Missing**	**Excluded**	**Coverage**
H:\Git\limits_verifier\src__init__.py	0	0	0	100%
H:\Git\limits_verifier\src\data_structures.py	3	0	0	100%
H:\Git\limits_verifier\src\errors.py	6	0	0	100%
H:\Git\limits_verifier\src\feedcode__init__.py	1	0	0	100%
H:\Git\limits_verifier\src\feedcode\api.py	24	0	0	100%
H:\Git\limits_verifier\src\feedcode\data_structures.py	10	0	0	100%
H:\Git\limits_verifier\src\feedcode\parser.py	35	0	0	100%
H:\Git\limits_verifier\src\feedcode\rules.py	15	0	0	100%
H:\Git\limits_verifier\src\file_paths.py	10	0	0	100%
H:\Git\limits_verifier\src\files__init__.py	3	0	0	100%
H:\Git\limits_verifier\src\files\breach_writer.py	28	0	0	100%
H:\Git\limits_verifier\src\files\error_writer.py	24	0	0	100%
H:\Git\limits_verifier\src\files\read_files.py	22	0	0	100%

H:\Git\limits_verifier\src\files\rules.py	2	0	0	100%
H:\Git\limits_verifier\src\limits__init__.py	3	0	0	100%
H:\Git\limits_verifier\src\limits\api.py	31	2	0	94%
H:\Git\limits_verifier\src\limits\data_structures.py	21	1	0	95%
H:\Git\limits_verifier\src\limits\limit_types.py	63	2	0	97%
H:\Git\limits_verifier\src\limits\parser.py	46	3	0	93%
H:\Git\limits_verifier\src\limits\rules.py	7	0	0	100%
H:\Git\limits_verifier\src\mi_log__init__.py	2	0	0	100%
H:\Git\limits_verifier\src\mi_log\api.py	39	2	0	95%
H:\Git\limits_verifier\src\mi_log\instance_collector.py	28	5	0	82%
H:\Git\limits_verifier\src\mi_log\parser.py	22	1	0	95%
H:\Git\limits_verifier\src\mi_log\rules.py	135	0	0	100%
H:\Git\limits_verifier\src\mi_log\utils.py	11	2	0	82%
H:\Git\limits_verifier\src\riskguard__init__.py	1	0	0	100%
H:\Git\limits_verifier\src\riskguard\api.py	66	3	0	95%
H:\Git\limits_verifier\src\riskguard\data_structures.py	7	0	0	100%
H:\Git\limits_verifier\src\riskguard\log_entry_handlers.py	98	6	0	94%
H:\Git\limits_verifier\src\riskguard\parser.py	44	0	0	100%
H:\Git\limits_verifier\src\riskguard\rules.py	42	0	0	100%
H:\Git\limits_verifier\src\rules.py	4	0	0	100%
H:\Git\limits_verifier\src\utils.py	11	0	0	100%
H:\Git\limits_verifier\src\verification__init__.py	1	0	0	100%
H:\Git\limits_verifier\src\verification\accountant.py	198	15	0	92%

H:\Git\limits_verifier\src\verification\data_structures.py	7	0	0	100%
H:\Git\limits_verifier\src\verification\ledger.py	204	8	0	96%
H:\Git\limits_verifier\src\verification\order_types__init__.py	2	0	0	100%
H:\Git\limits_verifier\src\verification\order_types\ledger_entry.py	34	0	0	100%
H:\Git\limits_verifier\src\verification\order_types\ledger_values.py	64	5	0	92%
H:\Git\limits_verifier\src\verification\order_types\ledger_variables.py	70	10	0	86%
H:\Git\limits_verifier\src\verification\order_types\outstanding_order.py	107	17	0	84%
H:\Git\limits_verifier\src\verification\order_types\outstanding_quotes.py	204	25	0	88%
H:\Git\limits_verifier\src\verification\ticker_tape.py	142	1	0	99%
H:\Git\limits_verifier\src\verification\verifier.py	24	0	0	100%

Appendix I. Original Project Description

About the project
Optiver has a lot of algorithms running which automatically or semi-automatically trade the global market. The programmers together with the risk department have included all kinds of limits in these algorithms. An example would be that a limit is set on the amount of orders being sent out to a particular exchange. So for example only 10 orders per second could be send out to one exchange.
Some of these limits can become very complicated, and can't be figured out if implemented correctly by the naked eye. We would like to get an insight if these limits are ever broken. The first part of the project involves the data analysis of our trades and positions thus far, such that we can see if any of the limits were broken in the past. The challenge lies into efficiently analyzing the data, as about a terabyte per day is generated in logs.
If this is finished in the 10 weeks time, we continue with a near-real-time insight into the broken limits. With this, we can check the same day if any of the algorithms makes a mistake.
The project will involve data analysis and solving the problem of making this insightful. The students will be left free to decide in what language to do it, and how to solve the problems. We will of course have an opinion about it, and are available to help in the process!

About the company
Over thirty years ago, Optiver started business as a single trader on the floor of Amsterdam's European Options Exchange. Today, we are a leading global electronic market maker, focused on pricing, execution and risk management. We provide liquidity to financial markets using our own capital, at our own risk, trading a wide range of products: from listed derivatives, cash equities, ETFs and bonds to foreign exchange. Our independence allows us to objectively improve the markets and provide efficiencies for market participants.
With 1000 Optiverians globally in Amsterdam, Chicago, Shanghai and Sydney our diverse team is united by our mission to improve the market. Thriving in a high performance environment, we pioneer our own trading strategies and systems using clean code and sophisticated technology. The technical challenges that this fast environment presents is what really motivates and challenges people at Optiver. To help us achieve what we want to attract, develop and empower top talent with a down to earth mentality, in order to sustain our future. What really sets Optiver apart from other companies is its openness, honesty and our colleagues. People have very strong opinions, but they are also open to listen and to change their mind if the facts presented by others have more value. The team shares the same goal, meaning that we are all pulling in the same direction and you as an individual care about how the person next to you is doing their job.

Additional information
The students who will be working on this project are Eric Cornelissen, Nick Winnubst, Joost Verbraeken and Cornel de Vroomen
Other information
The interns will receive an internship allowance of €500 gross per month.
This project requires the signing of a Non-Disclosure Agreement. If you have questions or concerns regarding this NDA, make sure to contact the company or the course coordinator BEFORE signing.