

Designing/implementing/evaluating an  
external map in Rust on modern hardware

# Agenda

- Introduction to the problem
- Data structures and their implementations
- Testing and profiling
- Experiments
- Conclusion

## Report

- 1 Introduction / Motivation
- 2 Problem formulation
- 3 Data structures
  - 3.1 Tables
  - 3.2 Binary Trees
    - 3.2.1 Binary Search Tree (BST)
    - 3.2.2 Red-black Tree
- 4 Design and Implementation
  - 4.1 Payload Map
  - 4.2 BST & Red-black Tree
  - 4.3 Table
  - 4.4 Why use Rust?
- 5 Running the program
- 6 Testing, Debugging, and Profiling
  - 6.1 Tests
    - 6.1.1 Unit tests
    - 6.1.2 Integration Tests
  - 6.2 Debugging
  - 6.3 Profiling
- 7 Experiments and Evaluation
  - 7.3 Experiment #1: Memory Usage
  - 7.4 Experiment #2: Caching
  - 7.5 Experiment #3: Search Time
  - 7.6 Experiment #4: Build Time

# Problem explained

- Siteimprove needs a webservice
- Primary focus is fast lookup time
- Run on cloud provider
- No need for persistent log

000.000.000.000	000.000.000.000	Payload
000.000.000.000	000.000.000.000	Payload
000.000.000.000	000.000.000.000	Payload
000.000.000.000	000.000.000.000	Payload
000.000.000.000	000.000.000.000	Payload

Example: "12.45.9.222 12.45.9.230 Siteimprove"

# Siteimprove's requirements

Language	Rust (or some other language Rust can call into).
Data set	A set of 150 million IP ranges with payload. IPs can be any IP in the full IPv4 range. No overlapping ranges. Up to 256bytes of payload pr. entry.
Pre-processing-time	Less than 24 hours. No new entries after initial pre-processing. No deletions or updates after initial pre-processing.
Disk usage	At most 100GB.
Lookup time	At most 40 milliseconds for average lookup time.
Memory	At most 4GB.

# Experience

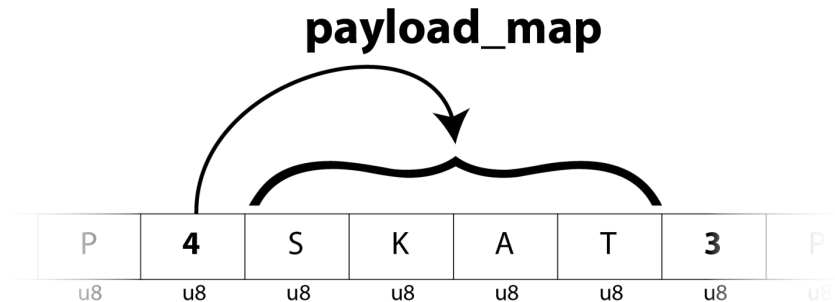
- Rust - First time using it
- Never worked with memory maps
- Never worked with cloud providers – e.g. DigitalOcean

# Data structures

- Binary search tree
- Red-black tree
- Table

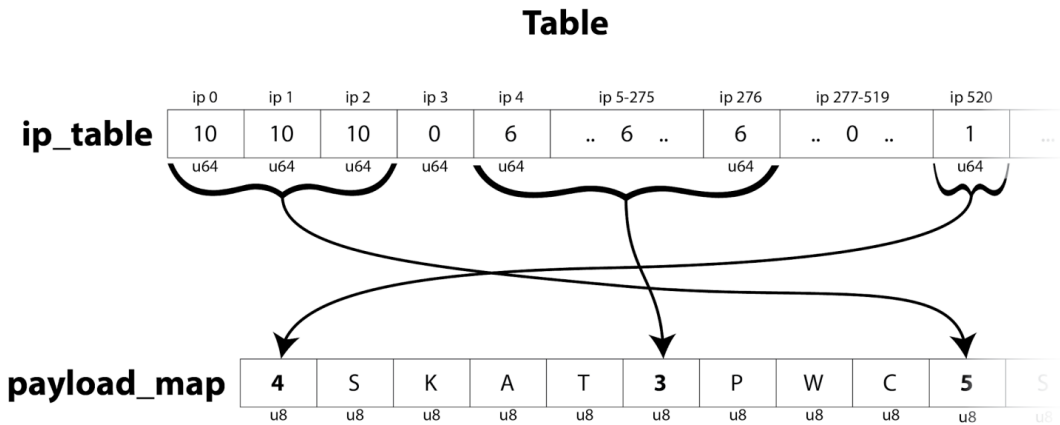
# Payload Map

- One memory mapped file with all payload
- Header to store the length
- New entries are appended at the end of the file
- Constant lookup time
- All data structures use it



# Table

- One memory mapped file with all possible IP addresses
- Each IP address contains a pointer to the payload in the payload\_map
- Constant lookup time
- Does not scale with number of entries



Lookup time complexity:  $O(1)$ .

Insert time complexity:  $O(r)$ , where  $r$  is the length of range of the entry.

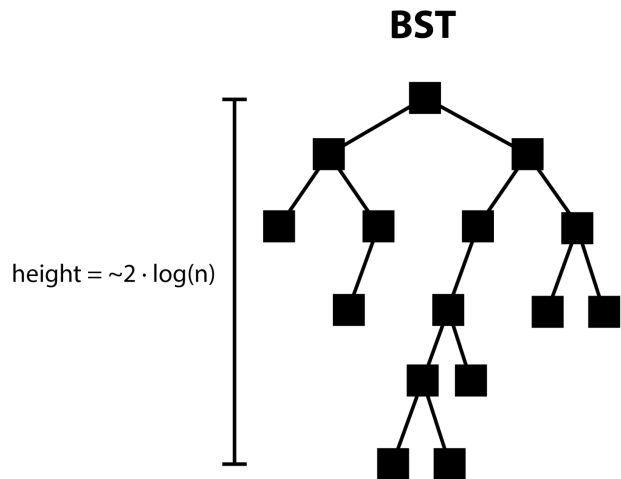
Space complexity:  $O(1)$

Space:  $2^{32} \cdot 64 \text{ bit} = 34.4 \text{ GB}$

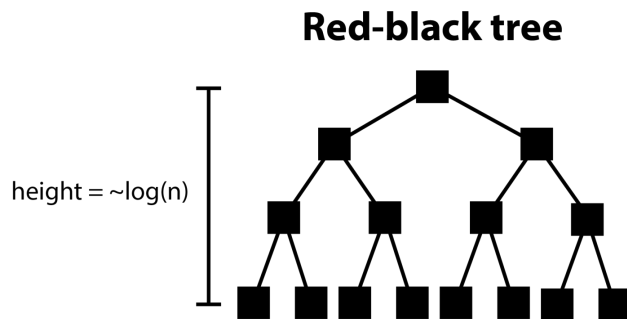


# BST vs. Red-black Tree

BST		
	Average	Worst case
lookup time	$O(\log(n))$	$O(n)$
Insert time	$O(\log(n))$	$O(n)$
Space	$O(n)$	$O(n)$



Red-black Tree		
	Average	Worst case
lookup time	$O(\log(n))$	$O(\log(n))$
Insert time	$O(\log(n))$	$O(\log(n))$
Space	$O(n)$	$O(n)$



# Trees: Memory Map structure

- Same structure
- Different root handling
- New nodes are appended at the end of the file

## BST

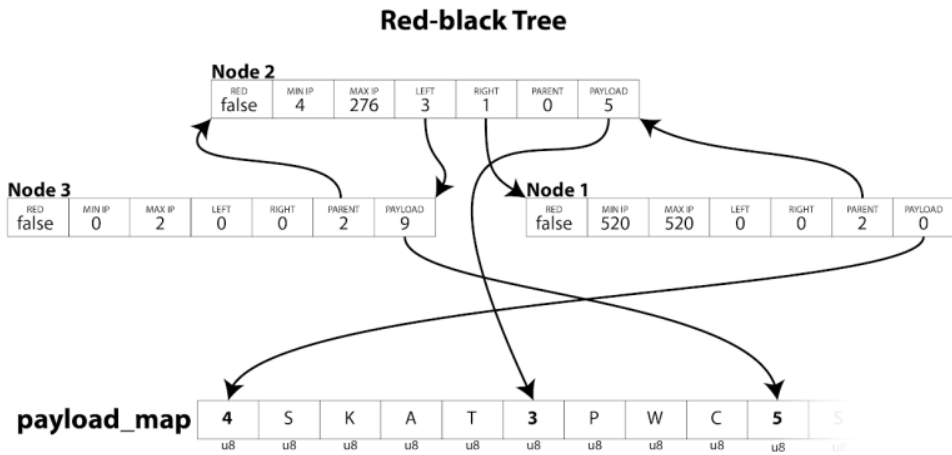
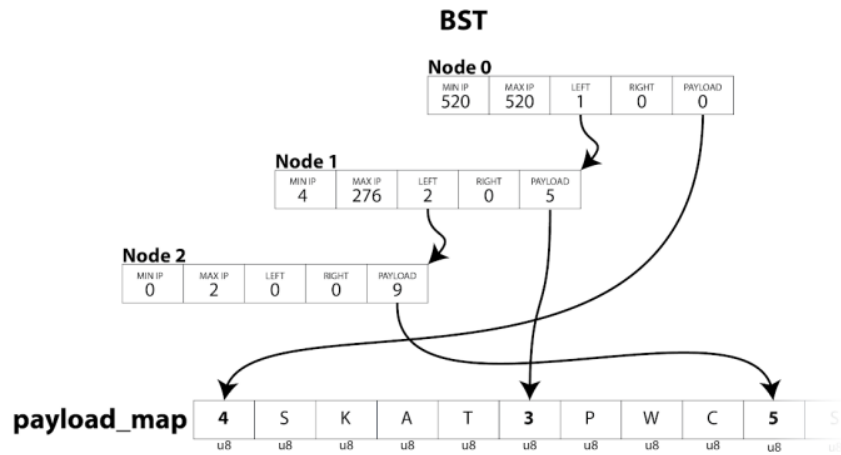
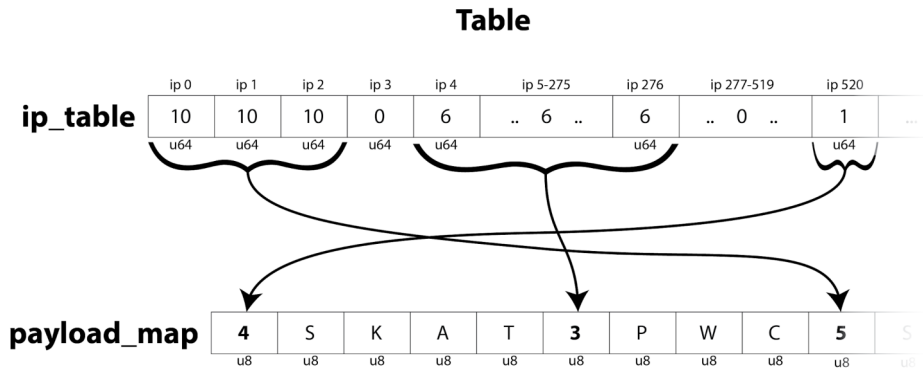
<b>Node 0 (Root)</b>	<b>Node 1</b>	<b>Node 2</b>	<b>Node 3</b>
----------------------	---------------	---------------	---------------

## Redblack Tree

Root node offset	padding	<b>Node 1</b>	<b>Node 2</b>	<b>Node 3</b>
---------------------	---------	---------------	---------------	---------------

# An example

```
0.0.2.8 0.0.2.8 SKAT
0.0.0.4 0.0.1.20 PWC
0.0.0.0 0.0.0.2 Siteimprove
```



# Testing

Unit tests

Integration tests

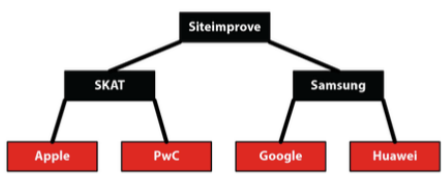
# Unit Testing

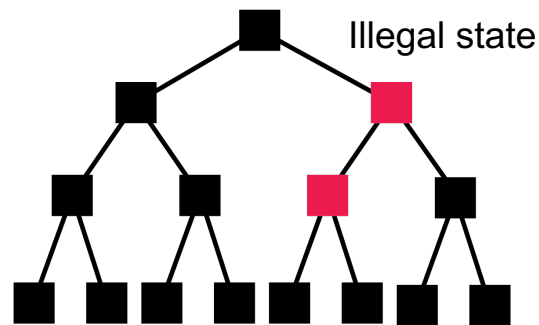
Almost all functions have tests

Positive and negative tests

Verifying that the red black tree was built correctly.

1. Printing the tree
2. Checking for corrupted family relations

Standard-output	Visual abstraction
<pre>-----X Huawei ---O Samsung -----X Google O Siteimprove -----X PwC ---O SKAT -----X Apple</pre>	



# Integrations tests

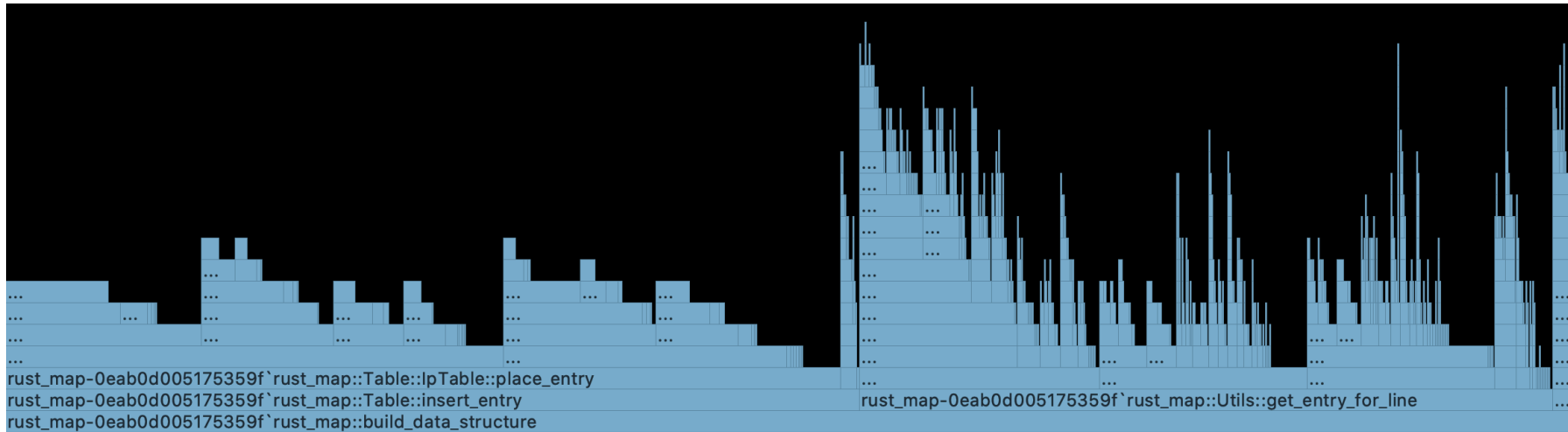
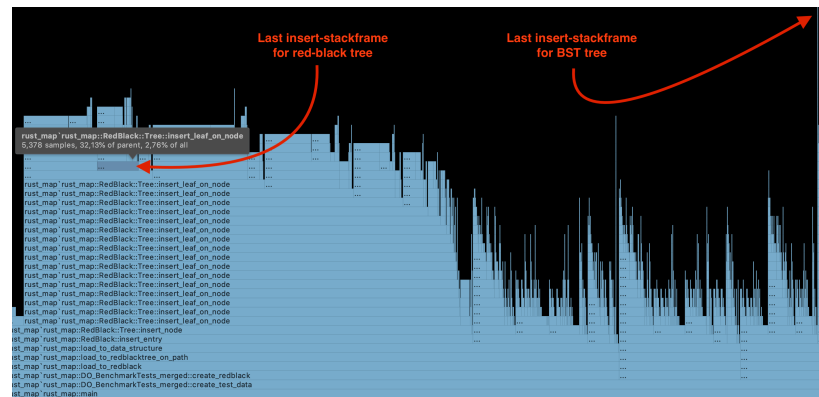
- Same steps as the experiments

# Profiler

- Flame Graph
- Call Tree

# Flame Graph

- Reading with regex
- Seeing the height of the trees
- Tail-end recursion





# Experiments

## Experiments

- Experiment #1: Memory Usage
- Experiment #2: Caching
- Experiment #3: Search Time
- Experiment #4: Building Time

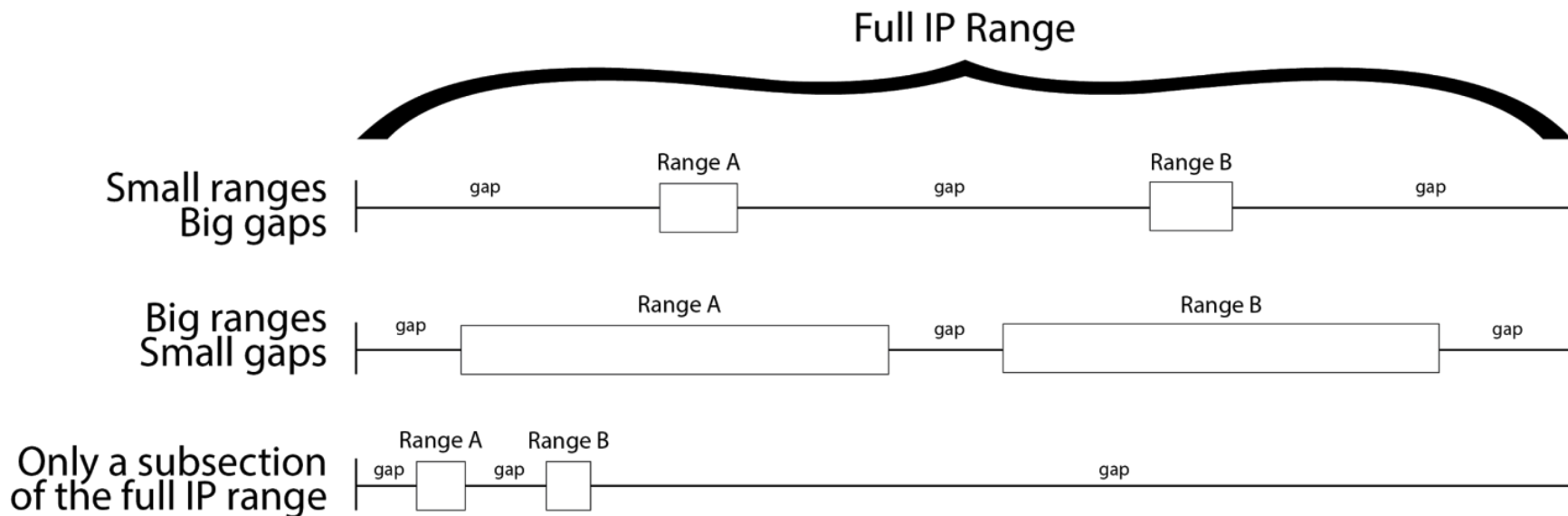
## Machines

- Dionysos: 128 GB memory machine.
- Droplet: 1 GB memory machine hosted on DigitalOcean.

## Data sets

- 150.000.000
- 10.000.000
- 100.000
- 1000

# Depends on how the data looks



# Experiment #1: Memory Usage

Expectation:

- Keeps loading in pages as long as there is free memory left

Discussion:

- Droplet started paging after 2% of the data was processed
- Dionysos loaded in everything and did not start paging

## Results

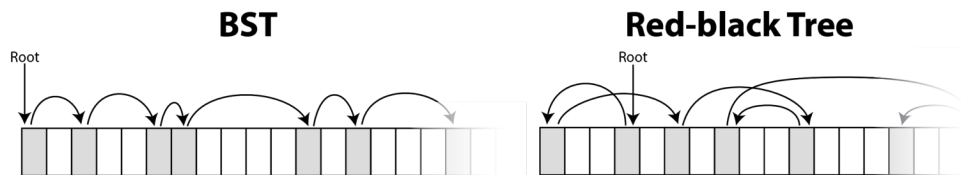
Memory Usage		
Machine	Droplet	Dionysos
Idle	106mb	9783mb
Running	168mb	9844mb
Difference	<b>62mb</b>	<b>61mb</b>

Page Cache		
Machine	Droplet	Dionysos
Idle	56mb	1371mg
Running	754mb	38709mb
Difference	<b>698mb</b>	<b>37337mb</b>

# Experiment #2: Caching

Expectation:

- Cache should not matter on infinitive large data sets
- Temporal Locality and Spatial Locality



Discussion:

- Difficult to isolate
- Dionysos 10mill and 150mill works as expected
- Inconsistent on Digital Ocean

## Results

Dionysos				
Structure	1k	100k	10mill	150mill
BST	36.1 %	54.7 %	61.4 %	68.2 %
Red-black	34.5 %	52.9 %	76.4 %	76.4 %
Table	35.1 %	79.9 %	83.7 %	78.9 %

Droplet				
Structure	1k	100k	10mill	150mill
BST	29.9 %	26.4 %	42.2 %	30.1 %
Red-black	40.4 %	29.2 %	25.0 %	42.3 %
Table	48.8 %	25.0 %	9.0 %	40.6 %

# Experiment #3: Search Time

## Results

Expectation:

- Table < Red-black Tree < BST

Discussion:

- Table is fastest on large data sizes
- Table is slowest on small data sizes
- The trees perform similarly
- Overhead on small data sizes
- Both machines perform evenly on small data sizes

Dionysos

Structure	1k	100k	10mill	150mil
BST	2.50	0.88	2.17	3.71
Red-black	2.78	1.00	1.75	3.25
Table	7.09	3.85	3.44	0.95

Droplet

Structure	1k	100k	10mil**	150mil**
BST	1.45	0.85	324,10	6731.96
Red-black	1.47	0.84	1284,22	7417.70
Table	5.52	4.52	2157.37	5997.26

# Experiment #4: Build Time

Expectation:

- Table is limited by write speed
- BST only writes twice
- Red-black tree is slow because it rotates

Discussion:

- All data structures run in less than a day
- The Red-black tree use 3.2 hours
- Table catches up to BST
- BST is fastest because of great use of locality and few writes.
- Red-black tree does not save nodes on the way down the tree

## Results







Build time per data structure

Structure	1k	100k	10mil	150mil
BST	3917	206485	29928726	835904134
Red-black	62624	6276430	647125934	11516763374
Table	21924	1202969	155465193	960806939

Average insertion time per entry per data structure

Structure	1k	100k	10mil	150mil
BST	3.91	2.06	2.99	5.57
Red-black	62.62	62.76	64.71	76.78
Table	21.92	12.03	15.55	6.41

# Living up to the goals

Language 	Rust (or some other language Rust can call into).
Data set 	A set of 150 million IP ranges with payload. IPs can be any IP in the full IPv4 range. No overlapping ranges. Up to 256bytes of payload pr. entry.
Pre-processing-time 	Less than 24 hours. Longest took 3.2 hours
Disk usage 	At most 100GB. Highest disk usage 72.8 GB
Lookup time 	At most 40 milliseconds for average lookup time. Slowest lookup time ~7.4 milliseconds
Memory 	At most 4GB. It ran on a 1GB Memory virtual machine on DigitalOcean

# Evaluation and conclusion

All tree structures live up to the goals.

I have not run the program on the real data

Either increase RAM for even faster performance - or save the hardware cost



# 35.000 IPv6 entries?

BST 

Red-black Tree 

Table 

$2^{128}$  u64-pointers =  $2.7 \cdot 10^{30}$ GB file

## My suggestion

- BST or Red-black Tree for 35k IPv6 entries
- Table for 150 million IPv4 entries

# Final thoughts

- I enjoyed it
- Never spent so much time on a project
- Learned more than I thought I would
  - especially all the non-programming related stuff

# Learnings

- Rust
- Memory Maps
- Cloud providers and SSH
- Automation scripts