
Parallelisierung des Wave Function Collapse Algorithmus in einem Kubernetes Cluster

Masterarbeit zur Erlangung des akademischen Grades
Master of Science
im Studiengang Digital Sciences
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von: Luca Stamos
Matrikel-Nr.: 11132237
Adresse: Weilerstr. 14
50765 Köln
luca_luca.stamos@mail.th-koeln.de

vorgelegt von: Stefan Steinhauer
Matrikel-Nr.: 11132517
Adresse: Am Kohlberg. 1
51643 Gummersbach
stefan.steinhauer@mail.th-koeln.de

eingereicht bei: Prof. Dr. Lutz Köhler
Zweitgutachter: Prof. Dr. Roman Majewski

Gummersbach, 26.06.2025

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Ort, Datum

Unterschrift

Kurzfassung / *Abstract*

Im Rahmen dieser Arbeit wurde eine verteilte Softwarelösung zur prozeduralen Generierung zweidimensionaler Landkarten auf Basis des Wave Function Collapse Algorithmus entwickelt. Ziel war es, die ursprünglich sequenzielle Arbeitsweise des Algorithmus durch eine gezielte Parallelisierung in einem Kubernetes-Cluster zu beschleunigen und skalierbar zu gestalten.

Die zentrale Idee besteht darin, Landkarten in Abschnitte aufzuteilen, die unter Berücksichtigung vorberechneter Ränder unabhängig voneinander berechnet werden können. Die Verteilung der Abschnitte erfolgt über eine Microservice-Architektur, die auf Technologien wie Python, RabbitMQ und MySQL basiert. Ein konfigurierbares Regelwerk und benutzerdefinierte Einschränkungen ermöglichen eine konsistente Generierung der Karteninhalte. Ein weiterer Faktor, der bei der Generierung eine besondere Rolle spielt, ist der eingeführte Parameter der Entropietoleranz, welcher einen hohen Einfluss auf die Diversität der generierten Landschaften hat.

Zur Evaluation wurden verschiedene Konfigurationen aus Kartengröße, Anzahl der Abschnitte, beteiligten Arbeitsmaschinen und Entropietoleranz untersucht. Die Ergebnisse zeigen, dass die verwendete Architektur eine signifikante Beschleunigung der Generierung ermöglicht – besonders bei großen Landkarten.

Die Arbeit belegt, dass der Wave Function Collapse Algorithmus unter geeigneten Bedingungen parallelisierbar ist und damit als Grundlage für skalierbare prozedurale Generierungssysteme dienen kann.

Vorwort

Das vorliegende Dokument wurde in Zusammenarbeit von uns, Luca Stamos und Stefan Steinhauer, als Abschlussarbeit des Masterstudiums im Bereich Digital Science verfasst. Auch wenn es sich hierbei um eine neue Thematik und andere Technologien handelt, stellt diese Arbeit in gewisser Weise eine Weiterentwicklung unserer Bachelorarbeit dar, die wir unter der Betreuung von Prof. Dr. Köhler und Prof. Dr. Majewski erfolgreich abgeschlossen haben. Beide Professoren haben uns nicht nur das Vertrauen geschenkt, dieses Projekt gemeinsam weiterzuführen, sondern uns auch die Möglichkeit gegeben, unsere Masterarbeit im Rahmen einer wissenschaftlich anspruchsvollen und praxisorientierten Forschungsarbeit zu realisieren. Die Entscheidung, diese Arbeit bei ihnen zu schreiben, hat uns die Chance gegeben, auf Grundlage unserer Bachelorarbeit neue wissenschaftliche Fragestellungen zu entwickeln und zu untersuchen. Des Weiteren möchten wir uns herzlich für die Bereitstellung des Arbeitsraumes und des Rechnernetzwerks bedanken, welches uns die nötige Infrastruktur zur Umsetzung unserer Arbeit zur Verfügung stellte. Ohne diese Ressource wäre die Durchführung der Arbeit in der vorliegenden Form nicht möglich gewesen. Besonders profitieren konnten wir von den Inhalten der Module *Verteilte Systeme/Parallelisierung* bei Prof. Dr. Köhler und *VDM/Kubernetes* bei Prof. Dr. Majewski, da diese Themen eine zentrale Rolle in dieser Arbeit spielen, und dort sowohl theoretische als auch praktische Kenntnisse vermittelt wurden. Ein weiterer Dank gebührt dem Künstler Aladin „Safric“ Harker, der uns die Bilder für unsere Landschaftselemente erstellt hat, wodurch die Anschaulichkeit unserer Landkarten deutlich erhöht werden konnte.

Wir möchten mit dieser Arbeit einen Beitrag zur Weiterentwicklung des Themenbereichs Parallelisierung, insbesondere sequenzieller Algorithmen, leisten und hoffen, dass unsere Ergebnisse möglicherweise einen Impuls für andere Studenten geben, auch in diesem Bereich zu forschen.

Abschließend bedanken wir uns bei allen, die uns während des gesamten Masterstudiums begleitet und unterstützt haben, sei es durch fachliche Beratung, Diskussionen oder persönliches Engagement.

Zuständigkeit

Die den einzelnen Bereichen zugeordnete Zuständigkeit kann der folgenden Tabelle entnommen werden. Punkte, die aufgrund der gemeinsamen Arbeitszeit nicht eindeutig zugeordnet werden können, sind in beiden Spalten gemeinsam aufgeführt. Zusätzlich möchten wir erklären, dass dieses Projekt in ständiger Kooperation entstanden ist und wir uns beide vollständig verantwortlich für die Arbeit fühlen.

Luca Stamos	Stefan Steinhauer
Wave Function Collapse Algorithmus Hub Service/Hub-DB Distributor Service/Parallelisierung Worker Service RabbitMQ Service Timekeeper Service/Time-DB Timeextractor Applikation Messreihe/Statistik Messtabellen/Auswertung Messungen Skript	Kubernetes Cluster/Architektur Manager Service/Rules Distributor Service/Parallelisierung Worker Service Maploader Applikation Messreihe/Durchführung Messtabellen/Auswertung Kubernetes Skripte Git Repository/Version Log

Inhaltsverzeichnis

Abbildungsverzeichnis	VII
Glossar	XII
1. Einleitung	1
1.1. Motivation	1
1.2. Ziel der Arbeit	1
1.3. Aufbau der Arbeit	2
2. Grundlagen	3
2.1. Wave Function Collapse Algorithmus	3
2.1.1. Funktionsweise	3
2.1.2. Anwendungsbereiche	8
2.2. Parallelisierung	8
2.2.1. Konzepte der Parallelisierung	8
2.3. Virtualisierung	9
2.4. Kubernetes	10
2.4.1. Container	11
2.4.2. Pods	13
2.4.3. Deployments	13
2.4.4. Cluster	13
2.4.5. Networking	15
2.4.6. Kubectl	15
2.4.7. Kubeadm	15
2.4.8. Kubelet	15
2.4.9. Etcd	15
2.5. Message-Queue	16
3. Konzept	17
3.1. Allgemein	17
3.2. Parameter	17
3.2.1. Maps	17
3.2.2. Chunks	22
3.2.3. Rules	23

3.2.4. Restrictions	23
3.2.5. Entropy Tolerance	25
3.3. Parallelisierung	28
3.4. Architektur	30
3.5. Evaluation	31
3.6. Technologien	32
3.6.1. Python	32
3.6.2. Python-Alpine	33
3.6.3. Kubernetes	33
3.6.4. RabbitMQ	33
3.6.5. MySQL	33
3.6.6. Oracle VirtualBox	34
4. Implementierung	35
4.1. Grundlegende Entscheidungen	35
4.1.1. Binäre Repräsentation	35
4.1.2. Service-Architektur	35
4.2. Wave Function Collapse Algorithmus	36
4.3. Services	45
4.3.1. Manager	45
4.3.2. Distributor	51
4.3.3. Hub	60
4.3.4. Hub-DB	65
4.3.5. RabbitMQ	67
4.3.6. Worker	69
4.3.7. Timekeeper	73
4.3.8. Time-DB	78
4.4. Visualisierung	80
4.4.1. Maploader	80
4.4.2. Visualization-Bibliothek	81
4.5. Kubernetes	85
4.5.1. Installation und Konfiguration	85
4.5.2. Deployment	88
4.6. Parallelisierung	98
4.7. Tools	98
4.7.1. Timeextractor	98
4.7.2. messen.py	101
4.7.3. messung.sh	108
4.7.4. Graphing Tool	110
4.7.5. build.sh	116
4.7.6. push.sh	116

4.7.7. rebirth.sh	116
4.7.8. rebuild.sh	116
4.7.9. bashrc	118
4.8. Problemlösungen	118
4.8.1. Broker vs. Message-Queue	118
4.8.2. Dockerfile Layer	119
4.8.3. Robustheit RabbitMQ	119
4.8.4. Robustheit Worker	120
4.8.5. Messungen per Skript	120
4.8.6. Invalide Messungen	120
4.8.7. Entropy Tolerance	121
5. Evaluation	122
5.1. Methodik und Aufbau	122
5.1.1. Verwendete Hardware	123
5.2. Durchführung	124
5.3. Messergebnisse	124
5.4. Analyse/Auswertung	126
5.5. Diskussion	136
5.6. Limitation	136
5.6.1. Distributor	136
5.6.2. Quadratische Maps	137
5.6.3. Monitoring	137
5.6.4. Aussagekraft der Daten	137
5.6.5. Parallelisierung	138
5.7. Beispiele	141
6. Fazit	149
6.1. Zusammenfassung	149
6.2. Ausblick	149
Literatur	151
A. Git-Repository	153
A.1. Git-Repository	153
A.2. VDM Dokumentation	153
A.3. Messergebnisse	153
A.4. Beispielbilder	153
B. Messreihen	154
B.1. Messreihen	154

Abbildungsverzeichnis

2.1.	Beispiel für den WFC-Algorithmus zur Generierung einer Karte von Räumen (Gumin 2025)	5
2.2.	Beispiel für den WFC-Algorithmus zur Generierung von Pflanzentexturen (Gumin 2025)	6
2.3.	Beispiel für den WFC-Algorithmus zur Generierung von geometrischen Mustern (Gumin 2025)	7
2.4.	Klassische Struktur vs. Container-Struktur (Kubernetes 2025g)	10
2.5.	Komposition von Containern (Baier 2017, S.38)	11
2.6.	Schichten eines Container-Dateisystems (Baier 2017, S.40)	12
2.7.	Kubernetes Cluster Komponenten (Kubernetes 2025a)	14
3.1.	Verwendetes Sprite-Set (Version 2 © Aladin „Safric“ Harker)	18
3.2.	Beispiel für Map mit Tileset-Version 2 und Kantenlänge 64	19
3.3.	Ausschnitt einer Map mit Tileset-Version 2	20
3.4.	Beispiel für Map mit Tileset-Version 1 und Kantenlänge 64	21
3.5.	Map, aufgeteilt in Chunks	22
3.6.	Rules-Tabelle	23
3.7.	Restrictions-Tabelle: Checkboxen	24
3.8.	Restrictions-Tabelle: Binärrepräsentation	24
3.9.	Map mit Entropietoleranz 0	26
3.10.	Map mit Entropietoleranz 5	27
3.11.	Software Architektur	30
4.1.	wave.py Imports	36
4.2.	wave.py get restrictions	37
4.3.	wave.py numberOfOnes()	37
4.4.	wave.py findLowestEntropyTile() 1	39
4.5.	wave.py findLowestEntropyTile() 2	39
4.6.	wave.py collapseTile()	40
4.7.	wave.py combinedTileCondition()	41
4.8.	wave.py updateMap()	42
4.9.	wave.py algorithmStep()	43
4.10.	Wave Function Lookup Table	43
4.11.	wave.py prettyPrintMap()	44

4.12. Beispiel einer Ausgabe mit prettyPrintMap()	44
4.13. Manager Dockerfile	46
4.14. Manager Template	46
4.15. Manager Homepage	47
4.16. Manager Imports und Read Rules	47
4.17. Manager Tile Lookup	48
4.18. Manager Routes 1	49
4.19. Manager Routes 2	50
4.20. Distributor Dockerfile	51
4.21. Distributor Imports	52
4.22. Distributor Routes	52
4.23. Distributor Template	53
4.24. Distributor Homepage	53
4.25. Distributor generateMap() 1	54
4.26. Distributor generateMap() 2	54
4.27. Distributor generateMap() 3	55
4.28. Distributor getRules()	55
4.29. Distributor setMap()	56
4.30. Distributor distributeMap() 1	57
4.31. Distributor distributeMap() 2	57
4.32. Beispiel einer vorberechneten Map mit 16 Parts	58
4.33. Beispiel einer vorberechneten Map mit 64 Parts	59
4.34. Hub Dockerfile	60
4.35. hub.py Imports	60
4.36. hub.py Global Variables	61
4.37. hub.py Route /saveChunk	62
4.38. hub.py Route /updateChunkByID	62
4.39. hub.py Route /saveChunks	63
4.40. hub.py Route /getMapChunkByChunkID	64
4.41. hub.py Route /getMapByMapID	64
4.42. Hub-DB Dockerfile	65
4.43. Hub-DB Init	66
4.44. RabbitMQ Management-Plugin Queues	67
4.45. RabbitMQ Management-Plugin Queued Messages	68
4.46. Worker Dockerfile	69
4.47. Worker callback() 1	70
4.48. Worker callback() 2	71
4.49. Worker Imports, URLs, Rabbit Connection	71
4.50. Worker main()	72
4.51. Worker sendChunkTimes()	72
4.52. Timekeeper Dockerfile	73

4.53. Timekeeper Imports und globale Variablen	74
4.54. Timekeeper Routes 1	75
4.55. Timekeeper Routes 2	75
4.56. Timekeeper Routes 3	76
4.57. Timekeeper Routes 4	77
4.58. Time-DB Dockerfile	78
4.59. Time-DB Init	79
4.60. Maploader Applikation Code	80
4.61. Maploader Applikation	81
4.62. Visualization Imports	81
4.63. Visualization initializeTiles()	82
4.64. Visualization selectImage()	83
4.65. Visualization showmap()	84
4.66. Kubernetes Install Script 1	86
4.67. Kubernetes Install Script 2	86
4.68. Kubernetes Install Script Manager	87
4.69. Custom Resources YAML	87
4.70. Deployment wfcdeploy.yaml 1	88
4.71. Deployment wfcdeploy.yaml 2	89
4.72. Deployment wfcdeploy.yaml 3a	89
4.73. Deployment wfcdeploy.yaml 3b	90
4.74. Deployment wfcdeploy.yaml 4	90
4.75. Deployment wfcdeploy.yaml 5	91
4.76. Deployment wfcdeploy.yaml 6	92
4.77. Deployment wfcdeploy.yaml 7	92
4.78. Deployment wfcdeploy.yaml 8	93
4.79. Deployment wfcdeploy.yaml 9	93
4.80. Deployment wfcdeploy.yaml 10	94
4.81. Deployment wfcdeploy.yaml 11	94
4.82. Deployment wfcdeploy.yaml 12	95
4.83. Deployment wfcdeploy.yaml 13	95
4.84. Deployment wfcdeploy.yaml 14	96
4.85. Deployment wfcdeploy.yaml 15	96
4.86. Deployment wfcdeploy.yaml 16	97
4.87. Deployment wfcdeploy.yaml 17	97
4.88. Timeextractor Imports	98
4.89. Timeextractor Export-Funktion 1	99
4.90. Timeextractor Export-Funktion 2	100
4.91. Timeextractor Main	100
4.92. messen.py Imports	102
4.93. messen.py Variablen	102

4.94. messen.py Main 1	103
4.95. messen.py Main 2	104
4.96. messen.py Main 3	105
4.97. messen.py Main 4	106
4.98. messen.py isEmpty()	106
4.99. messen.py Exit Funktion	107
4.100messung.sh Main 1	108
4.101messung.sh Main 2	108
4.102messung.sh Funktionen	109
4.103Graphing Imports	110
4.104Graphing Main	111
4.105Graphing findAndPlotRegression() 1	112
4.106Graphing findAndPlotRegression() 2	112
4.107Graphing plotExperimentData() 1	113
4.108Graphing plotExperimentData() 2	114
4.109Graphing plotExperimentData() 3	115
4.110Graphing plotExperimentData() 4	115
4.111build.sh Skript	116
4.112push.sh Skript	117
4.113rebirth.sh Skript	117
4.114rebuild.sh Skript	117
4.115bashrc Datei	118
 5.1. Beispielauszug der Datei messreihen.csv	122
5.2. Spezifikation der Mac Pro Computer	123
5.3. Ausschnitt 1 der Messreihentabelle	125
5.4. Unverteilte Berechnung, x-Achse: Kantenlänge in Tiles, y-Achse: gemessene Zeit in Sekunden	127
5.5. Unverteilte Berechnung, logarithmische Darstellung, x-Achse: Kantenlänge in Tiles, y-Achse: gemessene Zeit in Sekunden	128
5.6. Verteilung in 4 Parts	130
5.7. Verteilung in 16 Parts	131
5.8. Verteilung in 64 Parts	132
5.9. Verteilung in 256 Parts	133
5.10. Relation von Kantenlänge in Tiles zur gemessenen Zeit in Sekunden auf Basis der Chunkgröße	134
5.11. 16 Worker Vergleich	135
5.12. Beispiel einer Map, in der die Schnittkanten der Parallelisierung mit 64 Parts erkennbar sind	139
5.13. Beispiel einer Map, in der die Schnittkanten der Parallelisierung mit 4 Parts erkennbar sind	140

5.14. Beispiel einer Map mit Kantenlänge 16, unverteilt, mit einer Entropietoleranz von 5	141
5.15. Beispiel einer Map mit Kantenlänge 16, in 4 Parts, mit einer Entropietoleranz von 5	142
5.16. Beispiel einer Map mit Kantenlänge 64, in 4 Parts, mit einer Entropietoleranz von 1	143
5.17. Beispiel einer Map mit Kantenlänge 64, in 16 Parts, mit einer Entropietoleranz von 5	144
5.18. Beispiel einer Map mit Kantenlänge 128, in 16 Parts, mit einer Entropietoleranz von 0	145
5.19. Beispiel einer Map mit Kantenlänge 128, in 64 Parts, mit einer Entropietoleranz von 0	146
5.20. Beispiel einer Map mit Kantenlänge 256, in 64 Parts, mit einer Entropietoleranz von 1	147
5.21. Beispiel einer Map mit Kantenlänge 256, in 64 Parts, mit einer Entropietoleranz von 5	148
 B.1. Ausschnitt 1 der Messreihentabelle	155
B.2. Ausschnitt 2 der Messreihentabelle	156
B.3. Ausschnitt 3 der Messreihentabelle	157
B.4. Ausschnitt 4 der Messreihentabelle	158
B.5. Ausschnitt 5 der Messreihentabelle	159
B.6. Ausschnitt 6 der Messreihentabelle	160
B.7. Ausschnitt 7 der Messreihentabelle	161
B.8. Ausschnitt 8 der Messreihentabelle	162
B.9. Ausschnitt 9 der Messreihentabelle	163
B.10. Ausschnitt 10 der Messreihentabelle	164
B.11. Ausschnitt 11 der Messreihentabelle	165

Glossar

Chunk Abschnitt einer Map. 22

EntropyTolerance Die Entropietoleranz beim Erzeugen einer Map. 25

Map Eine durch den Algorithmus erzeugte quadratische Landkarte. 17

numberOfParts Anzahl der Abschnitte, in die eine Map zerlegt wird. 23

numberOfTiles Größe der zu erstellenden Map in Anzahl der Tiles als Tupel (X- und Y-Achse) . 23

numberOfWorkers Anzahl der zu verwendenden Worker. 23

Restrictions Bedingungen zur Auswahl von Tiles für die Felder einer Map. 23

Rules Festgelegte Regeln zum Erzeugen einer Map. 23

Sprite Ein Bild in standartisiertem Format, mit dem ein Tile befüllt wird. 17

Tile Ein Bild, welches ein Feld einer Map beschreibt. 17

WFC-Algorithmus Wave Function Collapse Algorithmus. 3

1. Einleitung

1.1. Motivation

Die prozedurale Generierung von Inhalten spielt bei vielen modernen Anwendungen eine zentrale Rolle. Insbesondere bei der Erstellung virtueller Welten, vor allem in der Spieleentwicklung, soll eine konsistente und zugleich vielfältige Darstellung von Umgebungen gewährleistet werden.

Ein Algorithmus, der in diesem Zusammenhang besondere Aufmerksamkeit erhält, ist der Wave Function Collapse Algorithmus (WFC). Dieser ermöglicht es, anhand von definierten Regelwerken lokal konsistente Muster zu erzeugen, die sich global zu strukturierten und logisch verbundenen Karten zusammensetzen lassen.

Die Motivation dieses Projekts liegt in der Entwicklung einer Möglichkeit, den WFC-Algorithmus zu parallelisieren, das heißt, die nötigen Berechnungen auf verschiedene Maschinen zu verteilen. Ziel ist es, eine Software zu schaffen, die performante Map-Generierung in einem Kubernetes-Cluster erlaubt und sich flexibel an unterschiedliche Anforderungen anpassen lässt.

1.2. Ziel der Arbeit

Das Ziel der Arbeit soll es sein, den Prozess mit Technologien gemäß des Industrie-standards zu implementieren, um realistische Einsetzbarkeit zu demonstrieren. Des Weiteren sollen zahlreiche Tests gemacht werden, um Effizienz, Skalierbarkeit und weitere Faktoren zu quantifizieren und zu evaluieren. Besonderer Fokus liegt dabei auf der Parallelisierung des WFC-Algorithmus, welcher ursprünglich auf sequenziellen Berechnungen basiert. Die besondere Herausforderung besteht darin, lokal erstellte Kartenausschnitte unabhängig voneinander zu generieren und sie anschließend zu einem konsistenten Gesamtergebnis zusammenzufügen.

1.3. Aufbau der Arbeit

Im ersten Kapitel wird zunächst auf die zugrunde liegende Problemstellung und das Ziel sowie den Aufbau dieser Arbeit eingegangen.

Anschließend werden im zweiten Kapitel die Grundlagen für die im Projekt wichtigen Themen behandelt. Dazu gehören neben dem WFC-Algorithmus auch Konzepte der Parallelisierung. Weitere Grundlagen bilden Informationen zu Kubernetes und Message-Queues.

Im dritten Kapitel wird das Konzept der Arbeit vorgestellt. Neben einer allgemeinen Erklärung des Projekts wird auch auf die Architektur und die verwendeten Technologien eingegangen. Zusätzlich wird die spezielle Verwendung des WFC-Algorithmus sowie der Parallelisierungsansatz erläutert. Zuletzt wird die geplante Evaluation beschrieben.

Das vierte Kapitel behandelt die Implementierung. Neben der konkreten Implementierung des WFC-Algorithmus wird die Funktionsweise der einzelnen Dienste und Anwendungen der Servicelandschaft erläutert und anhand von Code erklärt.

Im fünften Kapitel erfolgt die Evaluation. Aufbau sowie Durchführung der Messungen werden beschrieben und die Ergebnisse werden diskutiert. Zusätzlich erfolgt eine Bewertung der Parallelisierung und gegebener Limitationen.

Das sechste Kapitel bildet das Fazit, in dem eine Zusammenfassung des Projekts erfolgt und ein Ausblick auf Erweiterungsmöglichkeiten sowie zukünftige Forschungsansätze gegeben wird.

2. Grundlagen

2.1. Wave Function Collapse Algorithmus

Der Wave Function Collapse Algorithmus (WFC-Algorithmus) ist ein Verfahren zur prozeduralen Generierung von Inhalten, welches auf sogenannten *Constraints* (Bedingungen) basiert und von Prinzipien der Quantenmechanik, insbesondere der Superposition, inspiriert wurde. Der Algorithmus wurde im Jahr 2016 von Maxim Gumin (Gumin 2025) entwickelt und basiert auf der Idee, dass jedes Element innerhalb eines Feldes am Anfang in einer Überlagerung aller möglichen Zustände existiert. Man kann in diesem Zusammenhang von einem „Potenzial“ sprechen. Die Generierung erfolgt, indem rekursiv das am wenigsten determinierte Gitterelement (d. h. das mit der höchsten Entropie) ausgewählt und dessen Zustand festgelegt wird. Dies führt zu einer Kaskade von Einschränkungen (*Constraint-Propagation*) bei benachbarten Elementen – ein Prozess, der dem „Kollaps“ der Wellenfunktion in der Quantenphysik metaphorisch nachempfunden ist (vgl. Heaton 2025).

Ein verwandtes und häufig als theoretische Grundlage zitiertes Verfahren ist das Model-Synthesis-Verfahren von Merrell (Merrell 2021, Merrell 2022), das ähnlich wie der WFC-Algorithmus auf der Idee basiert, aus lokalen Einschränkungen global konsistente Strukturen zu erzeugen.

2.1.1. Funktionsweise

Zu Anfang des Algorithmus werden alle Potenziale auf eine Superposition aller möglichen Zustände gesetzt. Dementsprechend hat jedes Potenzial die Möglichkeit, zu jedem Zustand zu werden. Im nächsten Schritt wird zufällig ein erstes Potenzial ausgewählt und zu einem der möglichen Zustände festgelegt. Dieser Vorgang wird von hier an als das „Einstürzen“ oder das „Kollabieren“ eines Potenzials bezeichnet. Nach dem Einstürzen eines Potenzials werden im nächsten Schritt alle Potenziale, die mit dem neuen Zustand verbunden sind, beeinflusst. Dies geschieht durch die Restriktionen der Zustände. So müssen bei den verbundenen Potenzialen alle Zustände entfernt werden, welche eine der Restriktionen verletzen. Im nächsten Schritt wird erneut eines der Potenziale zur Einstürzung ausgewählt. Hierbei wird in Betracht gezogen, wie viele mögliche Zustände ein gegebenes Potenzial noch hat. Potenziale mit wenigen möglichen

Zuständen werden bei der Wahl des Potenzials zur Einstürzung bevorzugt, um zu vermeiden, dass es Potenziale gibt, welche keine möglichen Zustände mehr annehmen können. Die letzten beiden Schritte werden wiederholt, bis keine Potenziale mehr vorhanden sind und alle Zustände den Restriktionen entsprechend gewählt sind. Dieser Vorgang führt zu unterschiedlichen Ergebnissen der Zustände aufgrund der zufälligen Wahl der eingestürzten Potenziale. Jedoch folgen alle Ergebnisse den vorgeschriebenen Restriktionen.

Die Abbildungen 2.1-2.3 zeigen Beispiele für den Vorgang der Generierung einer Textur mit dem WFC-Algorithmus. Hier entspricht ein Pixel einem Potenzial und jede mögliche Farbe einem Zustand. Am Anfang sind alle Potenziale auf alle Zustände maximiert und erscheinen verschwommen. Je weniger Zustände ein Potenzial bzw. Pixel noch annehmen kann, umso definerter ist die Farbe des Pixels. Die gezeigten Vorgänge sind in nur je 9 Bilder aufgeteilt, während der Algorithmus selbst mehrere hundert Schritte durchlaufen hat und dementsprechend nur makroskopisch zu beobachten ist.

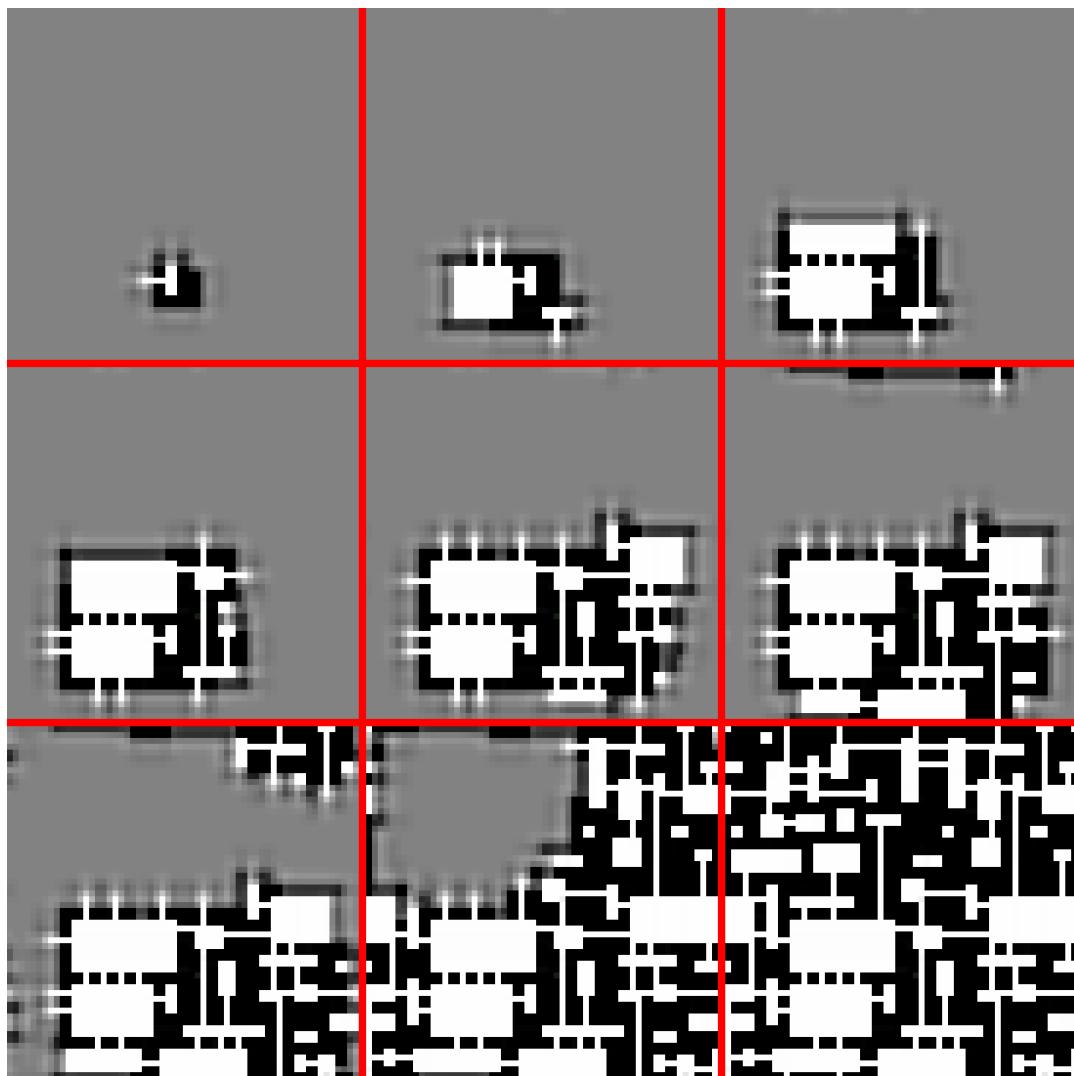


Abbildung 2.1.: Beispiel für den WFC-Algorithmus zur Generierung einer Karte von Räumen (Gumin 2025)

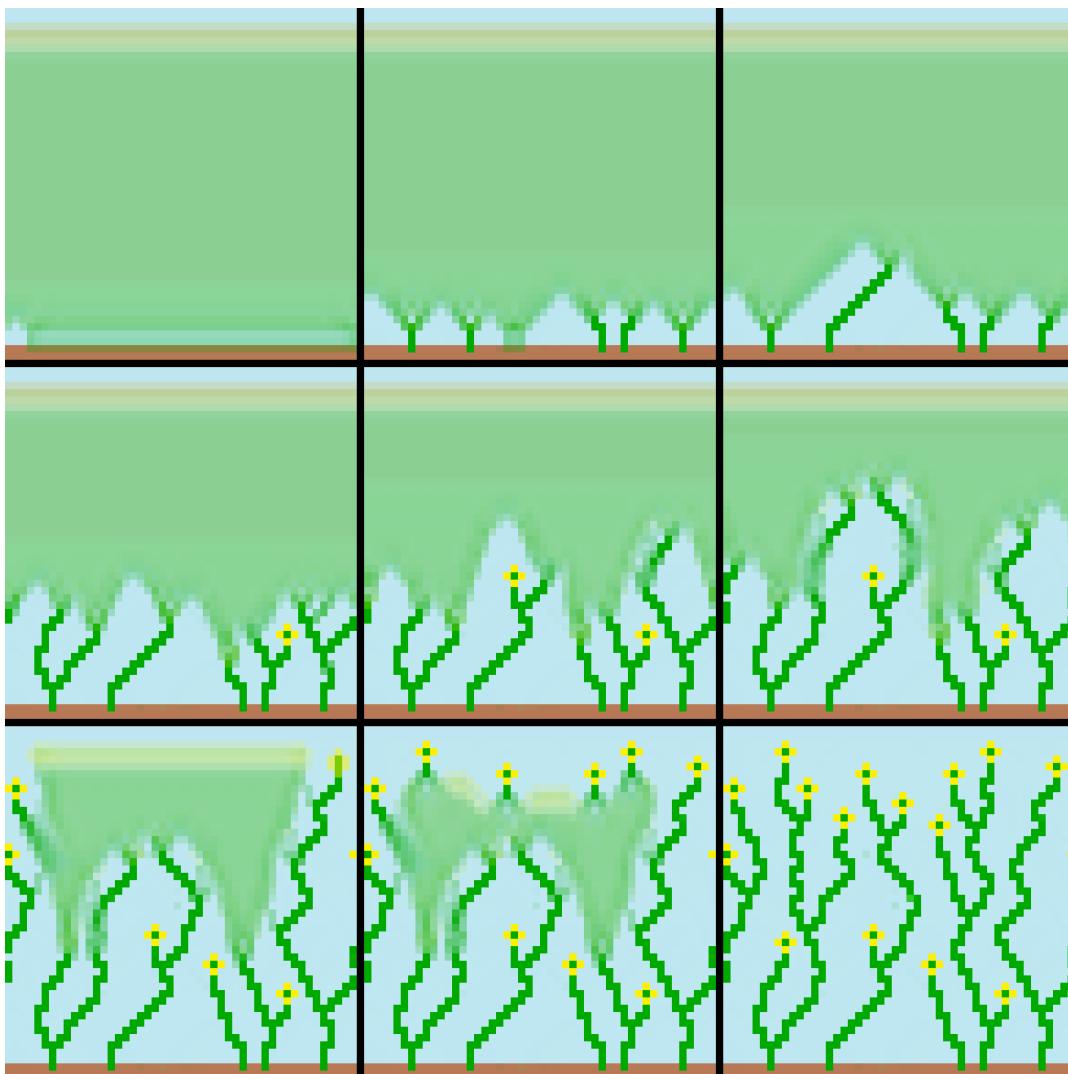


Abbildung 2.2.: Beispiel für den WFC-Algorithmus zur Generierung von Pflanzentexturen (Gumin 2025)

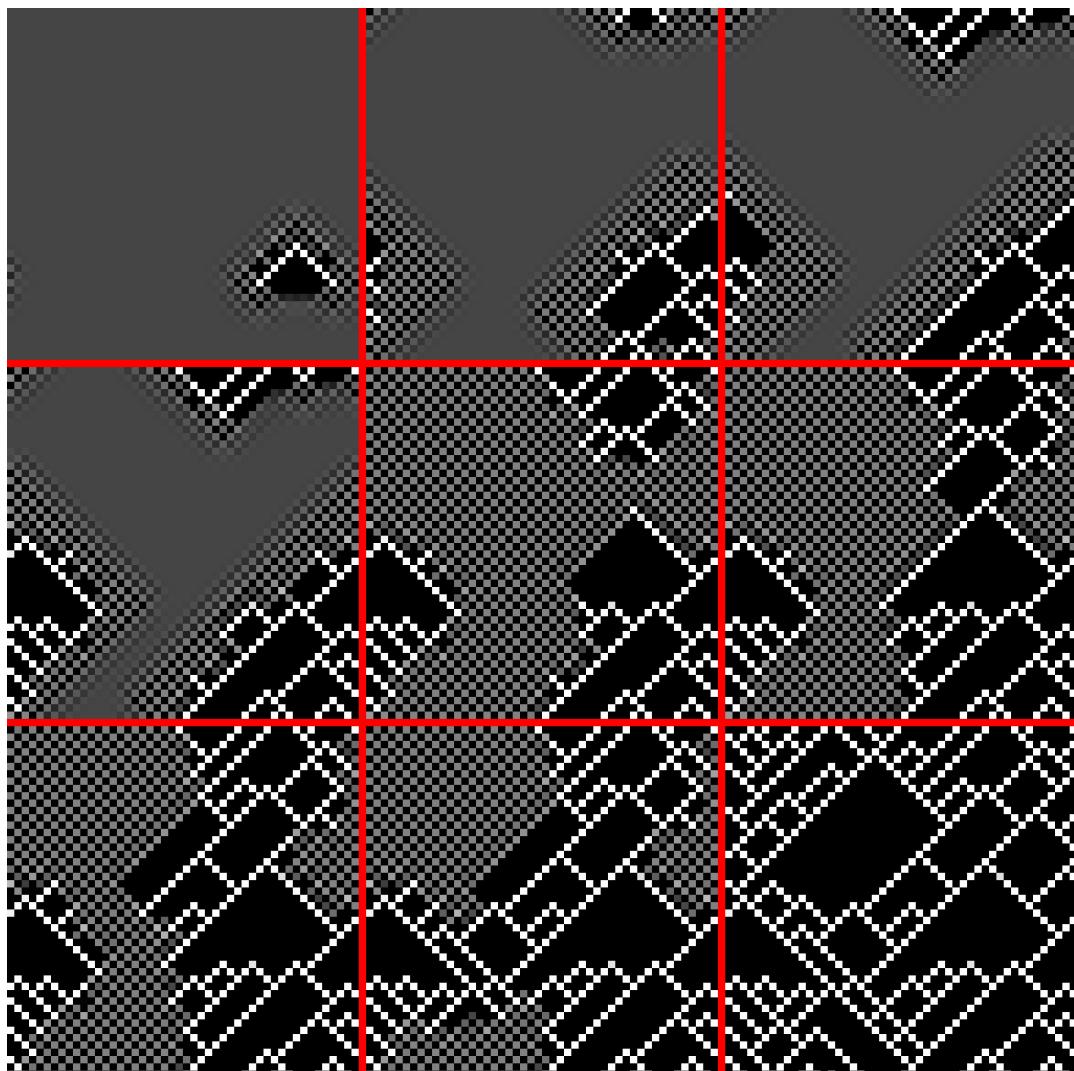


Abbildung 2.3.: Beispiel für den WFC-Algorithmus zur Generierung von geometrischen Mustern (Gumin 2025)

2.1.2. Anwendungsgebiete

Ursprünglich wurde der WFC-Algorithmus zur Generierung von zweidimensionalen Pixelmustern oder Kachelkarten eingesetzt, kann jedoch auch auf mehrdimensionale oder texturale Anwendungsfälle ausgeweitet werden. Der Algorithmus eignet sich besonders gut für die konsistente Generierung lokaler Strukturen, etwa bei der Synthese von Spielkarten, Terrain oder Texturen. Ein weiteres Beispiel für ein Anwendungsgebiet liefert Joseph Parker mit seinem Projekt „Generating Worlds With Wave Function Collapse“ in welchem er komplett Videospielwelten generiert (Parker 2025).

2.2. Parallelisierung

Parallelisierung ist der Vorgang des Aufteilens einer großen Aufgabe in mehrere kleinere Aufgaben, damit diese gleichzeitig abgearbeitet werden können. Der tatsächliche Aufwand der gesamten Aufgabe ändert sich hierbei nicht. In vielen Fällen werden bewusst größere Aufwandskosten in Kauf genommen, um Parallelisierung zu ermöglichen. Der Verlust lässt sich in vielen Fällen durch die Parallelisierung oft mehr als kompensieren. Dies liegt an der Eigenschaft des Computers, mehrere Aufgaben und Prozesse gleichzeitig abarbeiten zu können. Wird beispielsweise davon ausgegangen, dass ein Computer bis zu 4 verschiedene Aufgaben gleichzeitig abarbeiten kann, würde die Aufteilung einer Aufgabe theoretisch zu einem Performance-Anstieg von bis zu 300 Prozent führen. In der Realität werden mit spezieller Hardware, beispielsweise einer GPU (Graphics Processing Unit) weitaus mehr als 4 Prozesse gleichzeitig ausgeführt, oft mehr als 1000 (vgl. Gupta 2025). Je nach Problemstellung, Hardwarearchitektur und Softwaredesign können unterschiedliche Strategien zur Parallelisierung zum Einsatz kommen.

2.2.1. Konzepte der Parallelisierung

Formen der Parallelität

Ein zentrales Merkmal ist die Art der Parallelisierung. Man unterscheidet grundsätzlich zwischen dem *Datenparallelismus*, wo identische Operationen gleichzeitig auf unterschiedliche Daten angewendet werden, und dem *Aufgabenparallelismus*, in dem voneinander unabhängige Aufgaben gleichzeitig ausgeführt werden (vgl. Grama 2003, S.181ff und Mattson 2008, S.64).

Speicherarchitekturen

Die jeweils vorhandene Hardwarearchitektur bestimmt die Möglichkeiten der Parallelisierung maßgeblich. Grundsätzlich unterscheidet man zwischen einem *Shared-Memory-System*, in dem mehrere Threads auf denselben Speicher zugreifen, und einem *Distributed-Memory-System*, in dem verschiedenen Prozessen separate Speicherbereiche bereitgestellt werden (vgl. Mattson 2008, S.11f). Container-Orchestrierung mit Kubernetes verwendet z.B. ein Distributed-Memory-System.

Parallelisierungsmuster

Die Struktur für eine parallele Anwendung wird durch ein sogenanntes Parallelisierungsmuster beschrieben. Typische Strukturen sind:

- **Fork-Join:** Aufgaben werden in Teilaufgaben unterteilt (*Fork*) und nach Abschluss zusammengeführt (*Join*).
- **Pipeline:** Die Verarbeitung erfolgt in mehreren aufeinanderfolgenden Phasen mit möglicher Parallelisierung zwischen den Phasen.
- **Map-Reduce:** Daten werden parallel transformiert (*Map*) und anschließend aggregiert (*Reduce*).
- **Divide-and-Conquer:** Aufgaben werden rekursiv in kleinere Teilaufgaben zerlegt, welche parallel bearbeitet werden.

(vgl. Mattson 2008, S.167, S.73)

2.3. Virtualisierung

Virtualisierung bezeichnet das Erstellen von virtuellen Ressourcen. Dabei ist es mit Hilfe von Software möglich, eine Abstraktionsschicht über der physischen Computerhardware zu erstellen, mit der die Hardwareelemente eines Computers, wie Prozessor, Arbeitsspeicher oder Festplatten logisch in mehrere Computer - sogenannte *virtuelle Maschinen* (VMs) - unterteilt werden. Dabei bildet jede VM ihr eigenes, abgeschlossenes System mit eigenem Betriebssystem und isolierter Ausführung. Gängige Virtualisierungsplattformen sind unter anderem VirtualBox, Citrix Hypervisor oder Microsoft Hyper-V (vgl. IBM 2025b).

2.4. Kubernetes

Kubernetes (auch K8S) ist eine ursprünglich von Google entwickelte Open-Source-Plattform zur Bereitstellung und Verwaltung von containerisierten Anwendungen (vgl. Kubernetes 2025g). Dabei handelt es sich um ein Verfahren zur Virtualisierung minimalistischer Betriebssysteme, den sogenannten „Containern“, welche in der Lage sind, vordefinierte Programme auszuführen. Man spricht von Kubernetes in diesem Zusammenhang daher auch von einer Container-Orchestrierungssoftware (vgl. AWS 2025). Container-Anwendungen haben in den letzten Jahren deutlich an Beliebtheit gewonnen und wurden vor allem durch Docker revolutioniert (vgl. Baier 2017, S.37). Ein großer Vorteil dieses Verfahrens ist, dass alle Programme, die in den Containern ausgeführt werden, immer unabhängig von vorhandener Hardware auf die gleiche Weise ausgeführt werden können. Der Kompromiss besteht im zusätzlichen Aufwand oder „Overhead“, der mit den Betriebssystemen der Container einhergeht. Auch wenn diese möglichst klein gehalten sind, können sie trotzdem einen erheblichen Teil eines Containers einnehmen. Allerdings ist dieser Nachteil mit spezialisierter Hardware und dem generellen Anstieg von verfügbarem Speicher und Prozessorleistung verkraftbar. Somit hat sich Kubernetes zu einer der meistgenutzten Plattformen entwickelt und ist einer der Industriestandards im Bereich Cloud-Native-Entwicklung geworden (vgl. Baier 2017, S.389).

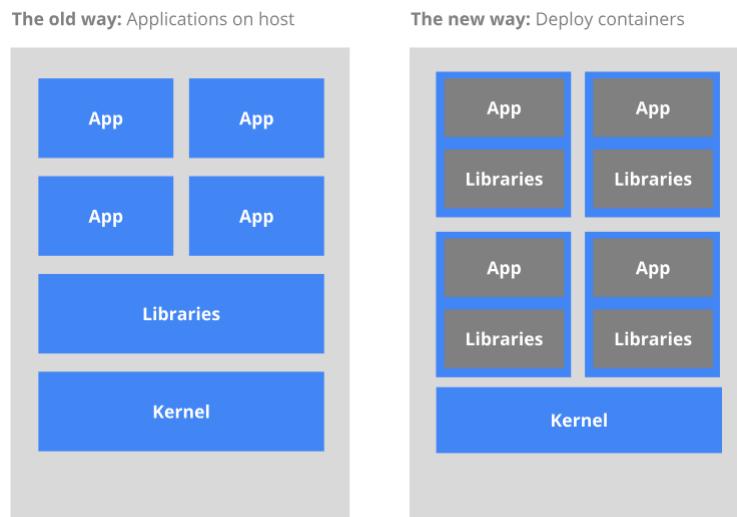


Abbildung 2.4.: Klassische Struktur vs. Container-Struktur (Kubernetes 2025g)

2.4.1. Container

Container sind eine Form der Virtualisierung auf Betriebssystemebene, bei der Anwendungen samt ihrer Abhängigkeiten in einer abgeschlossenen Einheit ausgeführt werden. Im Gegensatz zu allgemeinen virtuellen Maschinen (siehe Abschnitt 2.3) laufen Container parallel auf demselben Betriebssystem, wodurch sie besonders leichtgewichtig sowie einfach und schnell zu starten sind. Im Kubernetes-Umfeld stellen Container die grundlegende Ausführungseinheit dar. Kubernetes verwaltet sogenannte *Pods*, in denen ein oder mehrere Container laufen können. Eine Kerntechnologie bilden dabei die *Control Groups* (cgroups), welche den Zugriff auf vorhandene Hardware-Ressourcen regeln. Eine weitere Einheit sind *Namespaces*, welche für die logische Isolation zuständig sind. Sie trennen unter anderem Prozesslisten, Benutzer und Netzwerkschnittstellen, sodass jeder Container als eigenes System agieren kann (vgl. Baier 2017, S.38f). Das Dateisystem eines Containers setzt sich aus mehreren Schichten zusammen. Auf dem Basis-Kernel liegt das jeweilige Betriebssystem, auf dessen Basis wiederum die benötigte Software als zusätzliche Schicht angelegt werden kann (siehe Abbildung 2.5 und 2.6).

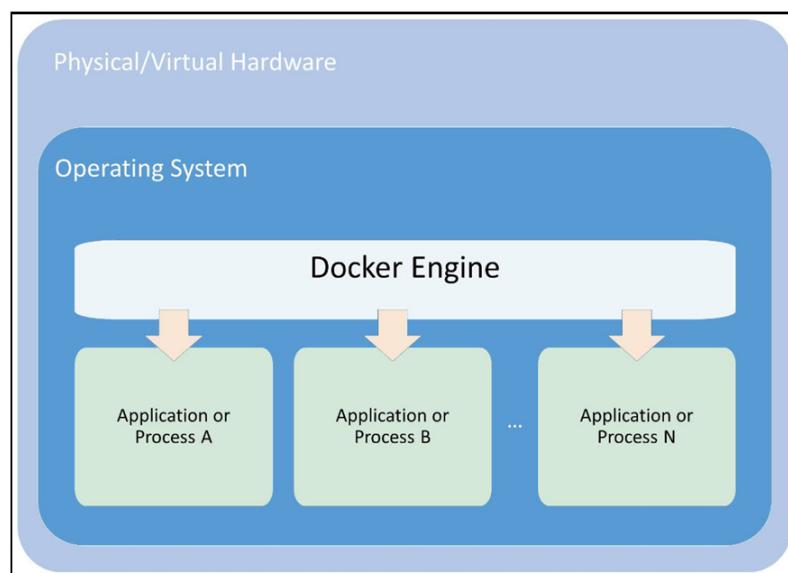


Abbildung 2.5.: Komposition von Containern (Baier 2017, S.38)

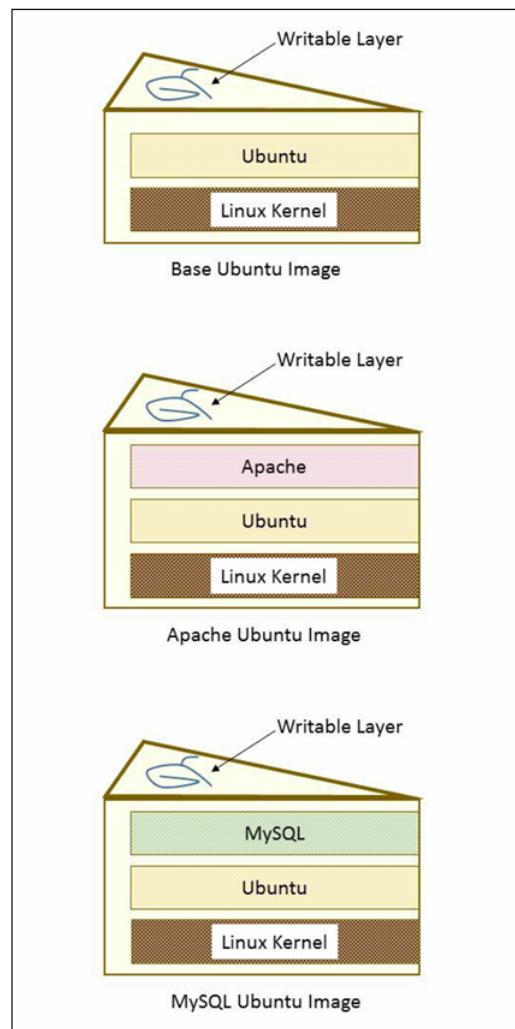


Abbildung 2.6.: Schichten eines Container-Dateisystems (Baier 2017, S.40)

2.4.2. Pods

Pods stellen in Kubernetes die kleinsten erstell- und verwaltbaren Einheiten dar. Sie beinhalten einen oder mehrere Container mit einer genauen Spezifikation und gemeinsam genutztem Speicher. Kubernetes unterstützt verschiedene Container-Umgebungen, wobei Docker die bekannteste ist und Pods mit der Docker-Terminologie beschrieben werden können (vgl. Kubernetes 2025f).

2.4.3. Deployments

Ein Deployment in Kubernetes ist eine Ressource, durch die bestimmt werden kann, wie viele Einheiten (*Pods*) einer Anwendung gestartet werden sollen. Man spricht in diesem Zusammenhang auch von *Replicas*. Dazu wird ebenfalls festgelegt, welches Image für den Container verwendet wird und auf welche Weise Updates ausgeführt werden. Dadurch wird von Kubernetes gewährleistet, dass die benötigte bzw. gewünschte Anzahl an Pods jederzeit verfügbar ist und dass ausgestorbene Pods automatisch ersetzt bzw. neu gestartet werden. Ein Deployment funktioniert *deklarativ*, das bedeutet, man beschreibt einen gewünschten Zustand und der Deployment-Controller ändert den aktuellen Zustand entsprechend (vgl. Kubernetes 2025b).

2.4.4. Cluster

Ein Kubernetes-Cluster beschreibt eine Gruppe von in einem Netzwerk verbundenen Maschinen, welche zusammenhängende größere Aufgaben verfolgen. Ein Cluster besteht immer aus mindestens einer Control-Plane und einem Knoten (Node) wobei die Control-Plane dafür verantwortlich ist, den gewünschten Zustand des Clusters zu erhalten, also welche Anwendungen gestartet, gestoppt oder skaliert werden. Die einzelnen Nodes werden dazu verwendet, Anwendungen in Betrieb zu nehmen und Workload (*Arbeitslast*) zu verteilen. Die Möglichkeit, Container auf verschiedenen Maschinen zu planen und auszuführen, bildet einen der Hauptvorteile von Kubernetes. K8S-Container sind nicht an einzelne Maschinen gebunden, sondern werden innerhalb des Clusters abstrahiert (vgl. RedHat 2025).

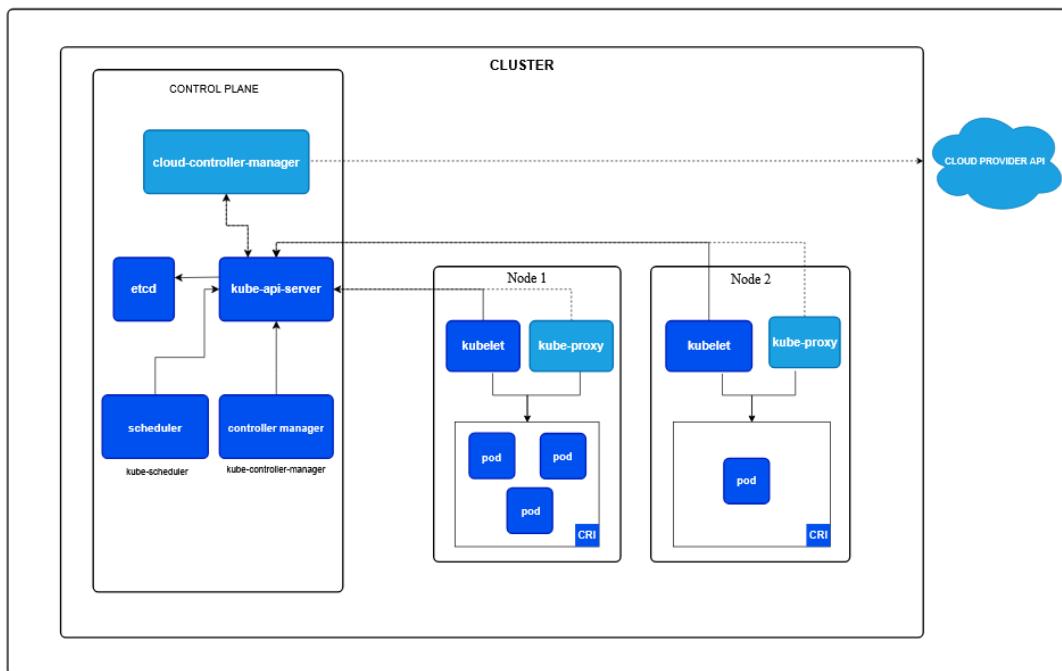


Abbildung 2.7.: Kubernetes Cluster Komponenten (Kubernetes 2025a)

2.4.5. Networking

Kubernetes bietet verschiedene Mechanismen, um eine Kommunikation zwischen Services sowie zum Cluster selbst zu konfigurieren. Während Funktionen wie *NodePort* oder *LoadBalancer* eine direkte Verbindung zu Pods herstellen können, ermöglicht das sogenannte *Ingress*-Netzwerk mithilfe eines Ingress-Controllers externen Zugriff auf die Dienste des Clusters (vgl. Baier 2017, S.146ff).

2.4.6. Kubectl

Kubectl ist ein Werkzeug von Kubernetes, welches zum Bereitstellen und Verwalten von Anwendungen innerhalb des Clusters verwendet wird. Es können Ressourcen sowie der Status der Nodes überprüft und Komponenten gestartet, gestoppt, gelöscht oder aktualisiert werden (vgl. Kubernetes 2025c und Hightower 2022, S.50).

2.4.7. Kubeadm

Kubernetes liefert mit dem Werkzeug Kubeadm eine Möglichkeit, den Cluster zu initialisieren (*init*) und Nodes dem Cluster hinzuzufügen (*join*), es bildet somit die Grundlage, ein Cluster aufzusetzen (vgl. Kubernetes 2025d).

2.4.8. Kubelet

Bei Kubelet handelt es sich um einen Agenten, welcher auf jeder Node ausgeführt wird und sicherstellt, dass Container in Kubernetes-konformen Pods ausgeführt werden. Er überträgt den Status der Applikationen sowie den Status der Node selbst über die API (vgl. Kubernetes 2025e und Lukša 2022, S.27).

2.4.9. Etcd

Das Etcd ist ein konsistenter und hochverfügbarer Key-Value-Speicher, welcher für die Sicherung aller für Kubernetes relevanten Daten des Clusters verwendet wird (vgl. Kubernetes 2025e und Lukša 2022, S.26).

2.5. Message-Queue

Eine Message-Queue (*MQ*) ist ein Mechanismus, welcher eine sogenannte „Warteschlange“ für Nachrichten implementiert. Eingehende Nachrichten werden in der MQ gespeichert, bis die konsumierende Anwendung bereit ist, die Nachricht zu empfangen. Es wird somit eine asynchrone Kommunikation zwischen verschiedenen Anwendungen oder Systemkomponenten ermöglicht (vgl. IBM 2025a). Üblicherweise sorgt eine Message-Queue dafür, dass eine Nachricht nur einmal von einem anderen Service gelesen und abgearbeitet wird. Darüber hinaus erhöht eine Message-Queue die Stabilität einer Software, indem sichergestellt wird, dass jede Nachricht abgearbeitet wird - auch dann, wenn eine vorherige Bearbeitung unerwartet abgebrochen wurde.

3. Konzept

3.1. Allgemein

Bei der in diesem Projekt entwickelten Software handelt es sich um eine Serviceland-schaft, in der zweidimensionale Landkarten (*Maps*) erstellt, gespeichert und abgerufen werden können. Die Generierung der Maps erfolgt mit Hilfe des Wave Function Collapse Algorithmus (WFC-Algorithmus). Die Verwaltung sowie die Kommunikation zwischen den einzelnen Diensten erfolgen innerhalb eines Kubernetes-Clusters. Es ist möglich, die Berechnung einer Map auf beliebig viele Arbeitsmaschinen (Worker-Nodes) zu verteilen, also zu parallelisieren, solange die entsprechende Map in genügend Abschnitte geteilt werden kann.

Der Wave Function Collapse Algorithmus ist eine Methode zur prozeduralen Generierung von Zuständen, welche durch Beziehungen untereinander verbunden sind (siehe auch Kapitel 2.1); in diesem Fall die Generierung einer Map, bei der die einzelnen Zustände als Bilder (Sprite) mit Umgebungsabbildungen realisiert sind. Die prozedurale Generierung dieser Zustände soll durch die Beziehungen der einzelnen Zustände untereinander beeinflusst sein. In diesem Fall wird dies benutzt, um Restriktionen zu setzen, welches Sprite neben welchem anderen Sprite generiert werden darf (siehe Abschnitt 3.2.4).

3.2. Parameter

3.2.1. Maps

Eine Map besteht aus einer Reihe, auf einem Raster angelegter Bilder (*Sprites*). Die Größe einer Map wird in ihrer Anzahl an Feldern angegeben und ist immer quadratisch. Ein Feld wird in diesem Zusammenhang als *Tile* bezeichnet. Eine Map der Größe X besteht also aus X Reihen je X Tiles (siehe auch Kapitel Limitation 5.6.2). Ein Tile kann eine von neun verschiedenen Formen annehmen, indem es mit einem der neun vorhandenen Sprites befüllt wird. Diese sind von dem durch den WFC-Algorithmus berechneten Zustand, an dieser Stelle der Map, abhängig. Abbildung 3.1 zeigt die neun verwendeten Sprites. Abbildung 3.2 - 3.4 zeigen Beispiele erzeugter Maps.



Abbildung 3.1.: Verwendetes Sprite-Set (Version 2 © Aladin „Safric“ Harker)

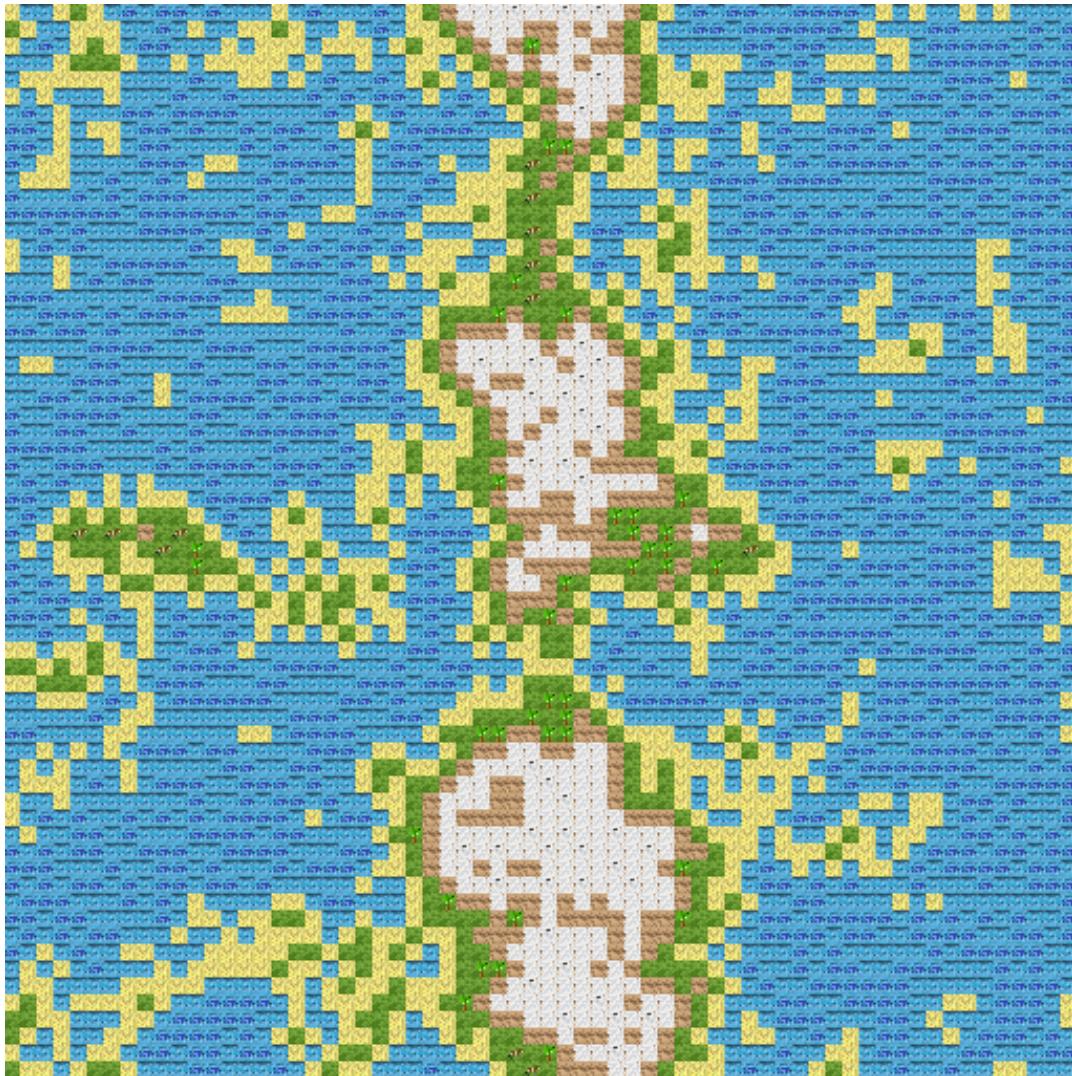


Abbildung 3.2.: Beispiel für Map mit Tileset-Version 2 und Kantenlänge 64

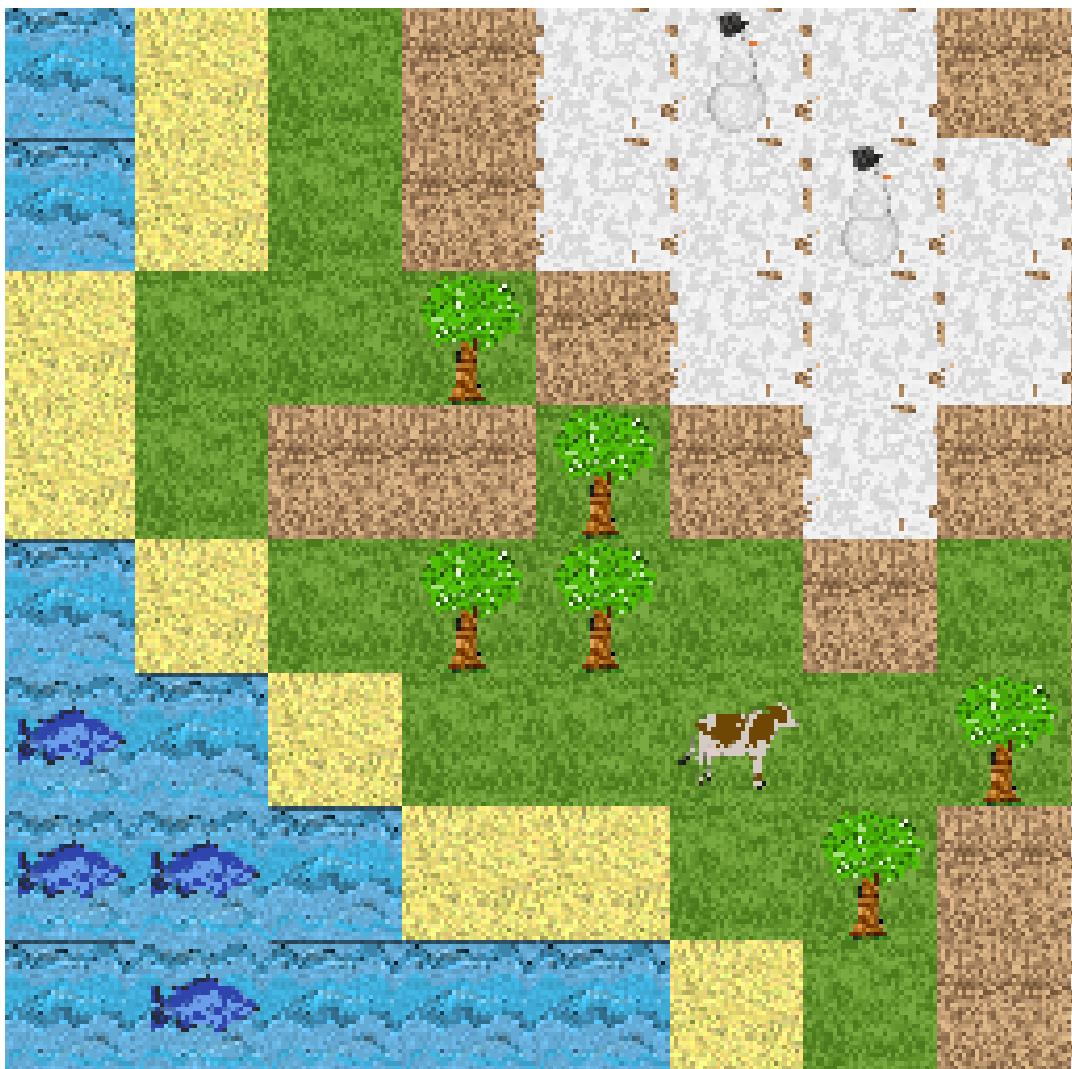


Abbildung 3.3.: Ausschnitt einer Map mit Tileset-Version 2

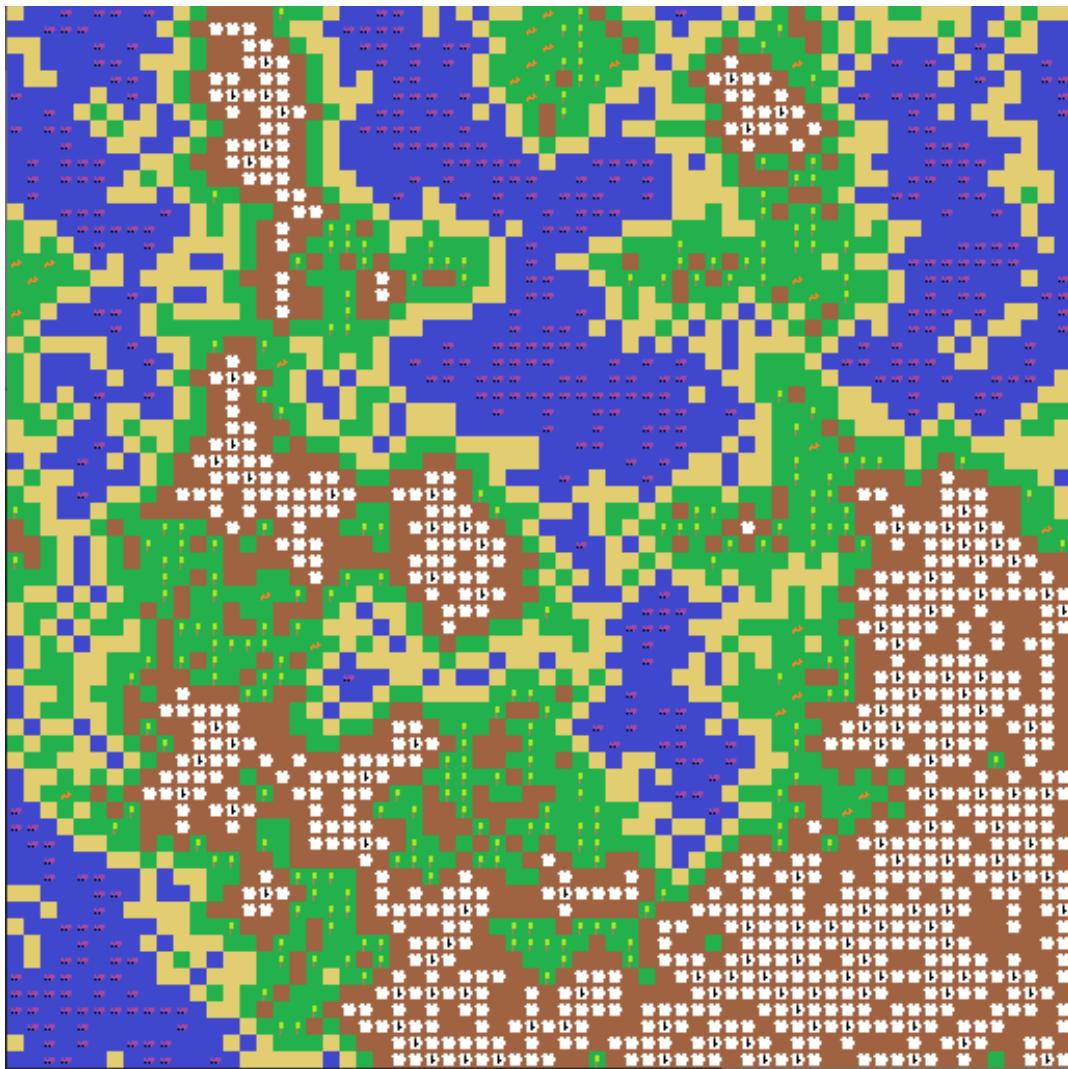


Abbildung 3.4.: Beispiel für Map mit Tileset-Version 1 und Kantenlänge 64

3.2.2. Chunks

Ein Chunk beschreibt einen durch Position und Größe definierten Abschnitt einer Map. Ein Chunk wird vor der Verteilung durch gezieltes Einstürzen der eingrenzenden Tiles begrenzt. Auf Grund dessen, dass Tiles durch dieses Einstürzen auch die angrenzenden Tiles beeinflussen, sind die Chunks von außen nach innen teilweise vorberechnet (siehe Abbildung 3.5). Eine Map kann nicht in beliebig viele Abschnitte geteilt werden, sondern ist an bestimmte Regeln gebunden (siehe Abschnitt Parallelisierung 3.3).

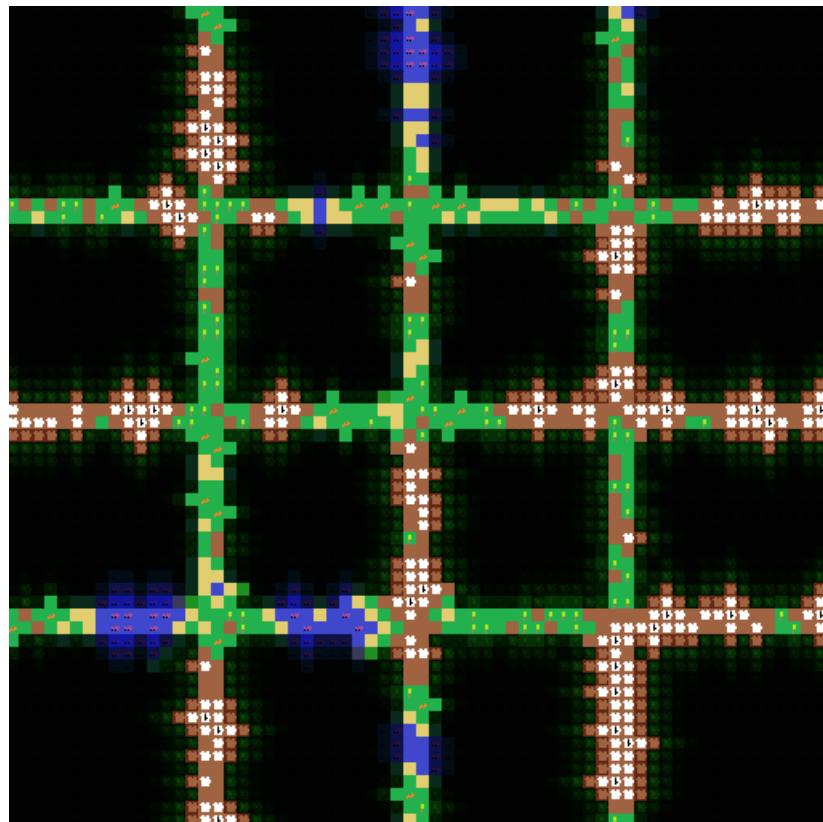


Abbildung 3.5.: Map, aufgeteilt in Chunks

3.2.3. Rules

Die Basis, auf der eine Map erzeugt wird, ist das Regelwerk (*Rules*). Dabei handelt es sich um eine Tabelle, in der alle nötigen Parameter zur Generierung gespeichert sind und die jederzeit geändert werden kann (Abbildung 3.6). Zu diesen Parametern gehören:

- **numberOfTiles**: die Größe der zu erstellenden Map in Anzahl der Tiles als Tupel (x- und y-Achse)
- **numberOfParts**: die Anzahl der Chunks, in welche die gesamte Map zerlegt werden soll (siehe auch Abschnitt Parallelisierung 3.3).
- **entropyTolerance**: Der Wert der Entropietoleranz beim Erstellen der Map (siehe auch Abschnitt 3.2.5)
- **numberOfWorkers**: die Anzahl der zu verwendenden Worker-Instanzen

	A	B
1	RULE	VALUE
2	numberOfTilesX	64
3	numberOfTilesY	64
4	numberOfParts	16
5	entropyTolerance	0
6	numberOfWorkers	2

Abbildung 3.6.: Rules-Tabelle

3.2.4. Restrictions

Um zu gewährleisten, dass der Aufbau bzw. der Inhalt einer generierten Map einem logischen Muster folgt und inhaltlich konsistente Ergebnisse liefert, gibt es spezielle Einschränkungen (*Restrictions*), welche Formen von Tiles nebeneinander liegen können. Diese Restrictions werden in einer dafür vorgesehenen Tabelle hinterlegt und können jederzeit geändert werden (Abbildung 3.7). Zusätzlich werden alle Restrictions in einer separaten Tabelle in Binärrepräsentationen von Integer-Werten zusammengefasst (Abbildung 3.8).

	Gras	Wald	Kuh	Strand	Wasser	Fisch	Berg	Schnee	Mann
Gras	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Wald	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Kuh	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Strand	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Wasser	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Fisch	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Berg	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Schnee	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
Mann	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>						

Abbildung 3.7.: Restrictions-Tabelle: Checkboxen

Binär	
Gras	0b001001111
Wald	0b001000011
Kuh	0b000000001
Strand	0b000011001
Wasser	0b000111000
Fisch	0b000110000
Berg	0b011000011
Schnee	0b111000000
Mann	0b010000000

Abbildung 3.8.: Restrictions-Tabelle: Binärrepräsentation

3.2.5. Entropy Tolerance

EntropyTolerance Entropy Tolerance (oder Entropietoleranz) ist ein optionaler Parameter für den Wave Function Collapse Algorithmus. Im normalen WFC-Algorithmus wird immer das Potenzial kollabiert, welches die niedrigste Anzahl an möglichen Zuständen (Entropie) hat. Auf diese Weise tendiert der Algorithmus jedoch dazu, homogenere Potenziallandschaften, oder in diesem Fall Maps, zu generieren. Möglicherweise liegt dies daran, dass das Tile, welches die niedrigste Entropie hat, immer neben einem bereits kollabierten Tile liegt. Aufgrund dessen, dass die Tiles in diesem Projekt nicht immer durch ein weiteres Tile verbunden werden können, entstehen „Fallen“ in den Landschaften. Diese führen dazu, dass ein „Biom“ auf der Map die Oberhand behält und am Ende sehr homogen aussieht (Abbildung 3.10 im Vergleich zu Abbildung 3.9). Für diesen Fall wurde die Entropietoleranz eingeführt. Diese erlaubt es dem Algorithmus auch, Tiles zu kollabieren, welche nicht die niedrigste Entropie auf der Map haben. Entropietoleranz beschreibt, wie hoch die maximale Differenz von der Entropie eines Tiles zur tatsächlich niedrigsten Entropie sein darf. Beispielsweise kann der Algorithmus mit Entropietoleranz 3 auch Tiles zum Einstürzen auswählen, welche eine Entropie von 5 haben, auch wenn die niedrigste vorhandene Entropie 2 ist. Durch diesen Eingriff in den Algorithmus ist dieser nicht mehr so stark dazu gezwungen, Tiles zu kollabieren, welche bereits kollabierte Nachbarn haben. Wie stark diese Abweichung jedoch sein sollte, ist eine subjektive Frage. Die in diesem Projekt generierten Maps tendierten dazu, bei höherer Entropietoleranz die Biome zu verkleinern. Die Wahrscheinlichkeit, mit der Generierung in ein anderes Biom überzugehen, ist erhöht. Doch scheint es sich bei sehr viel größeren Maps zu lohnen, die Entropietoleranz kleiner zu halten, weil diese dann detaillierter sind, während die Biom-Übergänge nicht so häufig vorkommen. Eine wichtige Anmerkung ist, dass diese Art von Abweichung des Algorithmus dazu führt, dass eine höhere Wahrscheinlichkeit besteht, eine fehlgeschlagene Generierung zu bekommen. Da nicht immer das Potenzial mit der niedrigsten Entropie kollabiert wird, kann es dazu kommen, dass ein Tile, welches direkt im nächsten Schritt kollabiert werden sollte, nicht kollabiert wird und es im schlimmsten Fall seinen letzten möglichen Zustand verliert. Dies kann mit dem originalen Algorithmus oder der gleichwertigen Entropietoleranz von 0 nicht geschehen.

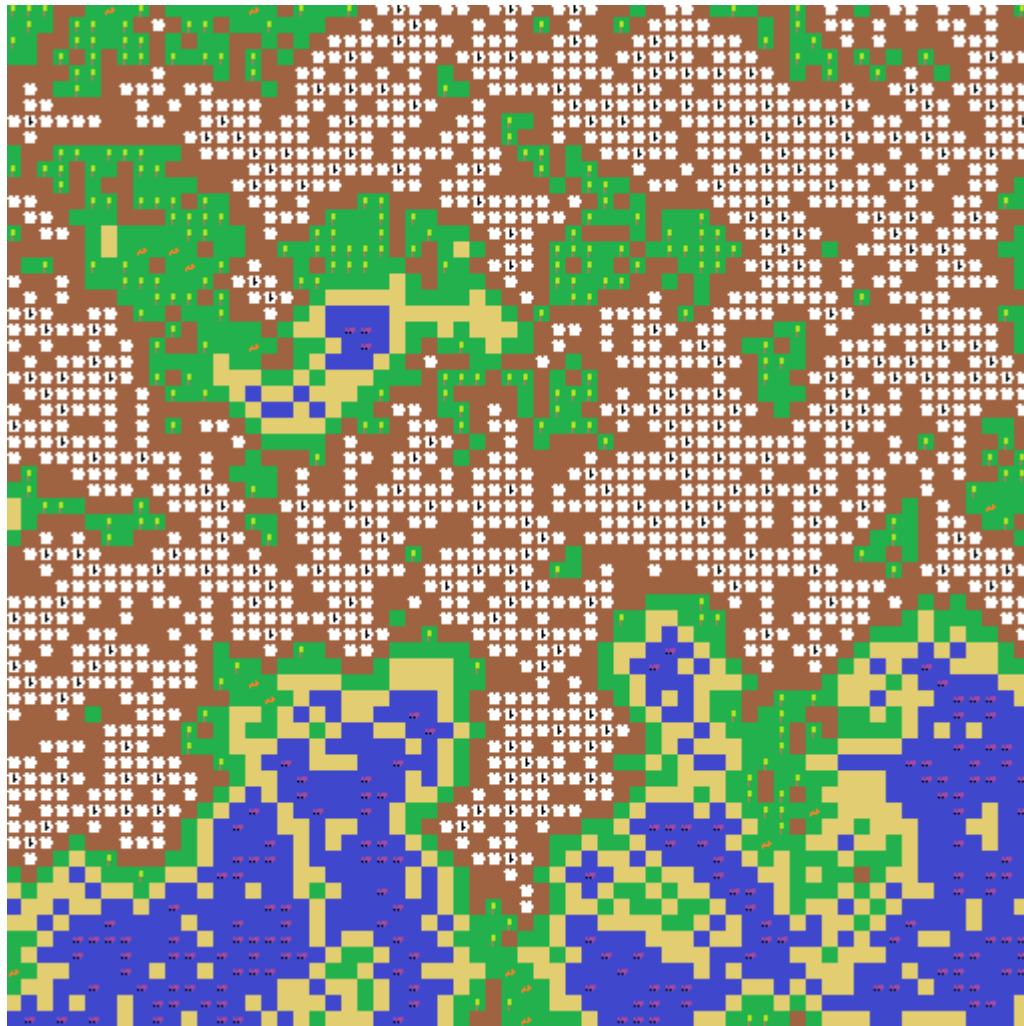


Abbildung 3.9.: Map mit Entropietoleranz 0

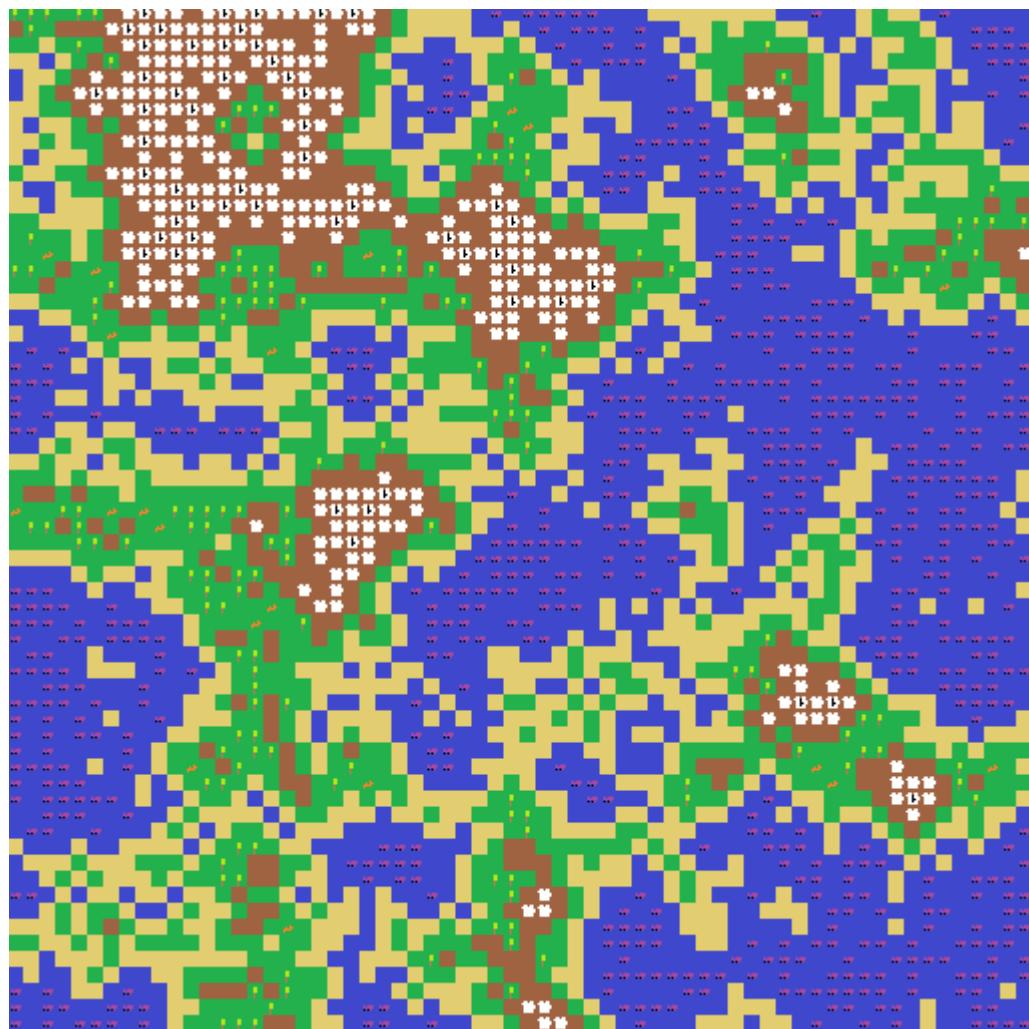


Abbildung 3.10.: Map mit Entropietoleranz 5

3.3. Parallelisierung

Auf Grund dessen, dass der Wave Function Collapse Algorithmus ein grundsätzlich sequenzieller Vorgang ist, ist die Aufgabe, diesen zu parallelisieren, keineswegs trivial. Die Grundidee der Lösung, die in dieser Arbeit beschrieben wird, ist es, den Algorithmus durch das Ausnutzen der Lokalisierung des Algorithmus zu parallelisieren. Grundsätzlich hat das Einstürzen eines Potenzials nur Auswirkungen auf andere benachbarte Potenziale, in diesem Fall die Tiles. Allerdings propagiert der Einsturz eines Potenzials durch die Gesamtheit aller Potenziale und ist somit nicht mehr lokalisiert.

Ein Versuch, die Potenziale einfach aufzuteilen und einzeln zu berechnen, würde aufgrund von nicht kompatiblen Einstürzungen von Potenzialen fehlschlagen, was zu Tiles führen kann, welche keinen möglichen Zustand (Sprite) mehr annehmen können. Dieses Propagieren kann jedoch durch das gezielte Einstürzen von angrenzenden Potenzialen eingegrenzt werden. Und somit kann die Lokalisierung des Algorithmus wiederhergestellt werden. So ist es möglich, einzelne Abschnitte zu erzeugen, welche unabhängig voneinander berechnet werden können. Dadurch können die Abschnitte im Weiteren auf mehrere Prozesse verteilt - und somit eine Parallelisierung erreicht werden. Dies geht allerdings mit verschiedenen Einschränkungen des Algorithmus einher, da hierdurch ein Schritt des Vorgangs, die Wahl des nächsten einzustürzenden Potenzials, stark abgeändert wird.

Die erste Einschränkung ist die Beschränkung, über wie viele Benachbarungen der Algorithmus andere Potenziale beschränken darf. Diese Einschränkung ist nicht unveränderlich, da eine Pufferzone in der Berandung der einzelnen Abschnitte eingeführt werden kann. Dies erhöht allerdings stark die Vorberechnungszeit, die für die Parallelisierung des Algorithmus notwendig ist, weshalb in diesem Projekt der Algorithmus auf direkt benachbarte Potenziale beschränkt wurde. Die nächste Einschränkung ist die oben erwähnte Vorberechnungszeit. Zu Beginn des Algorithmus ist somit ein sequenzieller Prozess notwendig, um die nachfolgende Parallelisierung zu ermöglichen. Der zusätzliche Aufwand sollte jedoch mit steigender Anzahl der Potenziale immer weniger Einfluss auf die Rechenzeit haben. In Kapitel 5 (Evaluation) wird genauer auf die Performance und diese Einschränkung eingegangen (siehe auch Abschnitt Limitation 5.6.1).

Vereinfacht zusammengefasst erfolgt die Parallelisierung durch das Einteilen einer Map in Abschnitte (Chunks). Bevor die einzelnen Abschnitte verteilt werden, wird der Rand jedes Chunks berechnet. So können die Chunks später anhand der Schnittkanten wieder zu einer gesamten Map zusammengesetzt werden. Ein Beispiel eines vorberechneten Randes einer Map ist im Abschnitt 3.2.2 in Abbildung 3.5 zu sehen.

Eine Map kann nicht in beliebig viele Chunks unterteilt werden. Die Parallelisierung kann nur unter bestimmten Voraussetzungen erfolgen. Diese lassen sich mathematisch wie folgt beschreiben: Sei eine quadratische Map mit der Kantenlänge k gegeben, so kann sie in jede beliebige Anzahl p Abschnitte geteilt werden, solange p das Quadrat einer natürlichen Zahl n - und p ein Teiler von k^2 ist. Es gilt:

$$p = n^2 \quad \text{mit} \quad n \in \mathbb{N} \quad \text{und} \quad p \mid k^2$$

3.4. Architektur

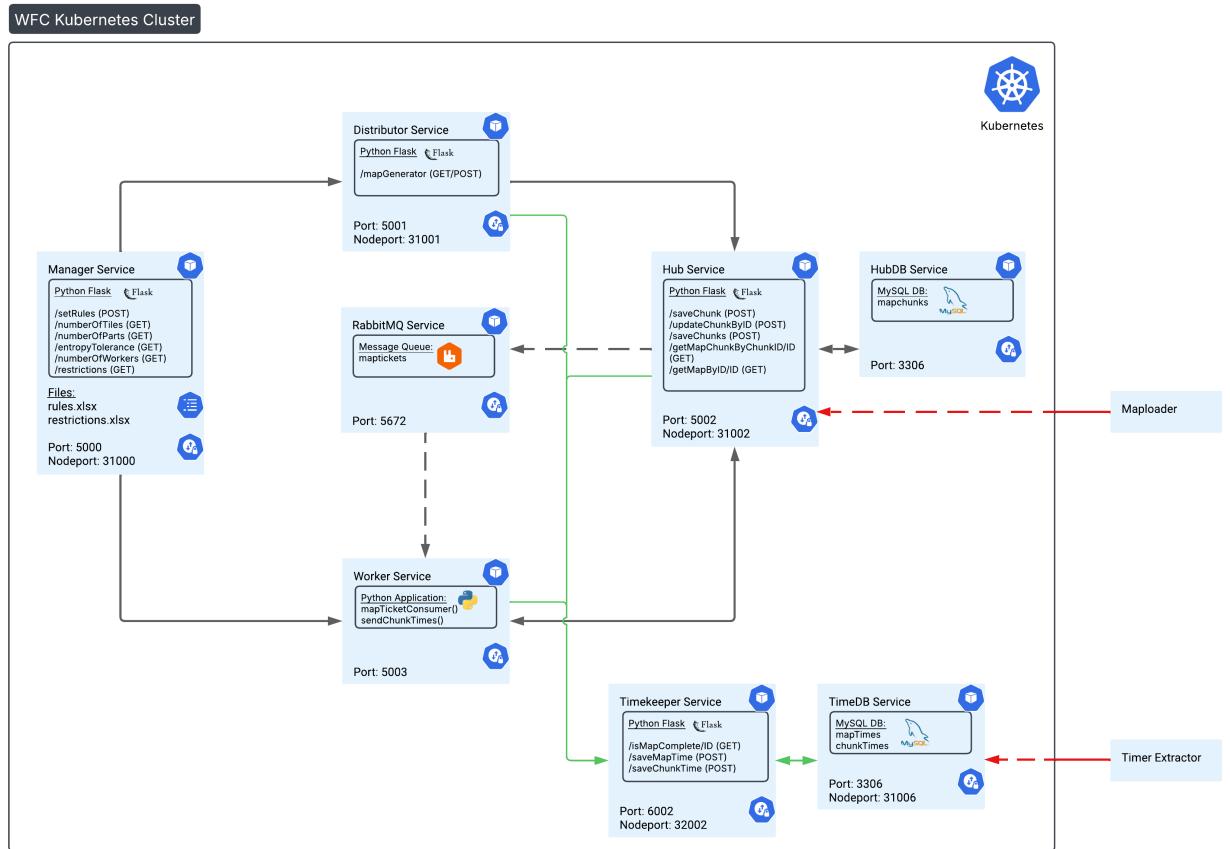


Abbildung 3.11.: Software Architektur

Die Abbildung (3.11) zeigt die gesamte Architektur der Software. Innerhalb des Kubernetes-Clusters existieren 8 Services:

- **Manager-Service**: Dienst zum Festlegen der Rules und Restrictions
- **Distributor-Service**: Dienst für die Vorberechnung der Ränder und zur Verteilung der Chunks
- **RabbitMQ-Service**: Message-Queue zur Verwaltung und Verteilung von Map-IDs an die Worker-Einheiten
- **Hub-Service**: Dienst zum Verwalten von Chunks und alleinige Kommunikation mit der Hub Datenbank

- **HubDB-Service:** Dienst für die dauerhafte Speicherung von Chunk-Informationen
- **Worker-Service:** Dienst zur Berechnung von Chunks
- **Timekeeper-Service:** Dienst zur Erfassung von Zeiten mit alleiniger Kommunikation zur TimeDB innerhalb des Clusters
- **TimeDB-Service:** Dienst zur dauerhaften Speicherung von Zeiten

Zusätzlich existieren außerhalb des Clusters zwei weitere Anwendungen:

- **Maploader:** Applikation zum Laden und zur grafischen Darstellung einer Map
- **Time Extractor:** Applikation zur Extraktion der Zeitdaten der TimeDB

Die genaue Funktionsweise der einzelnen Services und Applikationen wird in Kapitel 4 (Implementierung) im Detail erläutert.

3.5. Evaluation

Innerhalb des Projektes wird eine Reihe von Maps mit verschiedenen Parametern erzeugt. Zu diesen Parametern gehören neben der Größe der Map auch verschiedene Verteilungsoptionen wie die Anzahl der Chunks, in die eine Map zerlegt wird, und die Anzahl der beteiligten Worker.

Die benötigten Zeiten zum Generieren einer Map bzw. eines Chunks werden für jeden Vorgang zusammen mit den jeweiligen Parametern aufgezeichnet. So kann am Ende eine Evaluation der Messdaten durchgeführt werden, um Aussagen über optimale Verteilungsoptionen zu treffen und gegebenenfalls zu diskutieren. Um die Validität der Daten zu gewährleisten, wird jede Messung dreimal durchgeführt. Dies hilft bei der Erkennung von Ausreißern und liefert die Möglichkeit, einen Mittelwert der gemessenen Zeiten zu bilden. Die Erfassung der Messreihen wird auf einem Rechner-Cluster durchgeführt, bestehend aus 9 Computern, gestellt von der TH Köln, Campus Gummersbach.

3.6. Technologien

In diesem Abschnitt wird auf die innerhalb des Projektes verwendeten Technologien eingegangen und welche Rolle sie jeweils in der Software spielen.

3.6.1. Python

In diesem Projekt wurde sich aufgrund der Modularität und des geringen Overheads für die Programmiersprache Python entschieden, welche auf allen gängigen Betriebssystemen unterstützt wird. Zusätzlich existiert ein offizielles Docker-Image, welches bereits dazu optimiert ist, Python-Skripte auszuführen. Des Weiteren besteht um Python eine technisch versierte Community, welche diverse nützliche Module zur freien Nutzung bereitstellt.

Flask

Für die in diesem Projekt benötigten Aufgaben bietet Python ein minimalistisches und somit gut für dieses Projekt geeignetes Modul namens *Flask* an. Dieses lässt den Nutzer auf einfache Weise einen POST-Server aufsetzen, welcher als Backend für einen Service dient. Außerdem ist es gut gegen unvorhergesehene Fehler gewappnet und lässt sich aufgrund seiner Simplizität unkompliziert debuggen.

Pygame

Bei *Pygame* handelt es sich um eine Python-Bibliothek, welche grundsätzlich für die Entwicklung einfacher Videospiele gedacht ist. Aufgrund ihrer einfach gehaltenen visuellen Funktionen wird in diesem Projekt mit der Hilfe von Pygame die grafische Darstellung der Maps realisiert.

Pika

Pika ist die Python-Bibliothek, mit deren Hilfe das Python-Skript mit dem RabbitMQ-Service kommunizieren kann. Sie lässt das Programm Message-Queues eröffnen und konfigurieren. Hauptsächlich ermöglicht Pika es dem Python-Skript, als Message-Consumer zu fungieren, einzelne Messages zu erhalten und abzuarbeiten.

Pandas

Pandas ist eine Python-Bibliothek, welche grundsätzlich zur Analyse und Organisation von Daten verwendet wird. Dies beinhaltet auch die Möglichkeit, CSV-Dateien zu lesen und zu manipulieren. Sie wird in dieser Arbeit hauptsächlich für diesen Zweck verwendet.

3.6.2. Python-Alpine

Alpine ist ein minimalistisches Linux-Betriebssystem. Durch seine geringe Größe können sehr schlanke Docker-Images für Container erzeugt werden. Die Variante *Python-Alpine* zeichnet sich durch verschiedene vorinstallierte Python-Programme aus.

3.6.3. Kubernetes

Kubernetes ist eine Open-Source-Software zur Orchestrierung von containerisierten Anwendungen, welche Funktionen zur Verwaltung und Skalierung beinhaltet (siehe Kapitel 2.4).

3.6.4. RabbitMQ

RabbitMQ ist ein Message-Broker, der mehreren Services die Möglichkeit gibt, zu kooperieren, ohne Arbeit mehrfach oder gar nicht zu erledigen. Er bildet eine Nachrichtenwarteschlange (Message-Queue), bei der sich Services eine Nachricht abholen und abarbeiten können. Währenddessen kann kein anderer Service dieselbe Nachricht abholen. Wenn ein Service die Nachricht abgearbeitet hat, meldet er sich erneut bei dem RabbitMQ-Service und bestätigt die Komplettierung. So kann sichergestellt werden, dass jede Nachricht abgearbeitet wird und falls sich ein Service nicht erneut meldet, zum Beispiel, weil dieser abgestürzt ist, wird die Nachricht erneut zur Abholung bereitgestellt.

3.6.5. MySQL

„MySQL ist ein relationales Open-Source-Datenbankmanagementsystem (RDBMS), mit dem Daten gespeichert und verwaltet werden. Seine Zuverlässigkeit, Leistung, Skalierbarkeit und Benutzerfreundlichkeit machen MySQL zu einer beliebten Wahl

für Entwickler.“ (Erickson 2025). Die Daten der berechneten Chunks sowie die Zeitmessungen der Vorgänge werden jeweils in Datenbanken gespeichert. Hierzu werden innerhalb dieses Projekts MySQL-Datenbanken verwendet.

3.6.6. Oracle VirtualBox

In der ersten Phase der Entwicklung werden die Tests auf einem Verbund von virtuellen Maschinen durchgeführt. Die verwendete Host-Software dazu ist VirtualBox von Oracle.

4. Implementierung

In diesem Kapitel wird die Implementierung der innerhalb des Projekts entwickelten Software behandelt. Neben grundlegenden Entscheidungen werden die Funktionen der einzelnen Dienste und Applikationen sowie der verwendeten Skripte anhand von Code-Beispielen erläutert. Am Ende folgt ein Abschnitt, der sich mit einigen speziellen, während des Projekts aufgetretenen Problemen und Lösungen befasst.

4.1. Grundlegende Entscheidungen

4.1.1. Binäre Repräsentation

In diesem Projekt wurde sich dafür entschieden, die einzelnen Potenziale als eine Liste an True/False-Werten zu implementieren. Um die Performance und Speichereffizienz zu erhöhen, wird diese Liste als ein einzelner Integer-Wert implementiert, in dem jedes Bit des Integers einen Zustand repräsentiert. So können Bedingungen und Berechnungen durch einfache binäre Rechenoperationen erfolgen, ohne List-Comprehension anwenden zu müssen.

4.1.2. Service-Architektur

Es wurde sich dazu entschieden, die Funktionalität der Software in verschiedene Services zu unterteilen. Die Arbeit mit Kubernetes bietet einige Vorteile im Sinne der Robustheit, Skalierbarkeit und Kompatibilität. Um diese Vorteile effektiv nutzen zu können, schien diese Architektur als die sinnvollste. Es wurde sich grundsätzlich an einer Microservice-Architektur mit loser Kopplung zwischen den Diensten orientiert, während gleichzeitig versucht wurde, zu vermeiden, dass das Projekt zu komplex und unhandlich wird. Deshalb wurden an einigen Stellen leichte Kompromisse eingegangen, wodurch zusätzlich die Netzwerkzeit geringer gehalten werden kann.

Ein weiterer Punkt, welcher die Architektur betrifft, ist die Trennung von Datenverwaltung und Datenspeicherung. So laufen Datenbanken immer in einem eigenen Service. Dadurch wird einerseits Datenredundanz vermieden und zusätzlich wird der Zugriff

von außerhalb des Clusters durch die Verwendung von Node-Ports stark vereinfacht. Eine Darstellung der Architektur ist in Kapitel 3.4 zu sehen.

4.2. Wave Function Collapse Algorithmus

Die grundsätzliche Funktionsweise des Wave Function Collapse Algorithmus ist in Kapitel 2.1 beschrieben. Im folgenden Abschnitt wird auf die konkrete Implementierung innerhalb dieses Projekts eingegangen.

Imports

Abbildung 4.1 zeigt die für diesen Algorithmus genutzten Bibliotheken. Bei den drei importierten Bibliotheken sind zwei extern und eine intern. *Random* dient zur Generierung von pseudozufälligen Zahlen, welche im Algorithmus benötigt werden. Die externe *Time*-Bibliothek wird benötigt, um das Programm an bestimmten Stellen anzuhalten, für den Fall, dass eine Verbindung nicht hergestellt werden kann. Letztlich wird die eigens entwickelte Bibliothek *wavefunctionlookup* (Abbildung 4.10, siehe auch Abschnitt 4.2) benutzt, um die Zusammenhänge und Restriktionen der zuvor festgelegten Tiles abzurufen.

```
1 import random
2 import wavefunctionlookup as wfl
3 import time
```

Abbildung 4.1.: wave.py Imports

Get Restrictions

In Abbildung 4.2 zu sehen sind die für den WFC-Algorithmus benötigten globalen Variablen und deren erstmalige Zuweisung. Zusätzlich werden die für den WFC-Algorithmus benötigten Restriktionen durch die *wavefunctionlookup*-Bibliothek (4.2) von einem anderen Service abgefragt. Sollte diese Verbindung fehlgeschlagen, wartet der Algorithmus 60 Sekunden vor einem weiteren Versuch.

```

5   # GET RESTRICTIONS
6   while True:
7       try:
8           wfl.requestRestrictions()
9           break
10      except:
11          print("Connection Failed")
12          time.sleep(60)
13
14  numberOfTiles = (-1,-1)
15  entropyTolerance = -1
16  map = []

```

Abbildung 4.2.: wave.py get restrictions

Number of Ones

In Abbildung 4.3 ist eine nützliche Funktion für den WFC-Algorithmus zu sehen. Das Zählen der Einsen in der binären Repräsentation eines Integer-Wertes (Vgl. Robotbugs 2025). Hier wird zuerst von einer Anzahl von Null ausgegangen. Als nächstes beginnt eine Schleife, welche unterbricht, sobald der übergebene Integer n eine 0 enthält, was gleichwertig ist mit dem Wert False . Sollte dies nicht der Fall sein, wird die Variable für die Anzahl der Einsen inkrementiert. Als nächstes wird eine binäre *UND*-Operation mit den Werten n und $n-1$ ausgeführt. Dies führt dazu, dass die in der Binärrepräsentation am weitesten rechts angeordnete 1 auf 0 gesetzt wird. Dieser Schritt wird wiederholt, bis die Variable n den Wert 0 hat. Nun entspricht die Anzahl der Durchgänge der Anzahl der Einsen in der Binärrepräsentation der Variablen n , welche in der Variablen c gespeichert ist. Im letzten Schritt wird die Variable c von der Funktion zurückgegeben.

```

19  def numberOfOnes(n):
20      c = 0
21      while n:
22          c += 1
23          n &= n - 1
24  return c

```

Abbildung 4.3.: wave.py numberOfOnes()

Find Lowest Entropy Tile

In den Abbildungen 4.4 und 4.5 ist die Funktion zu sehen, welche aus der Liste der Potenziale (Tiles) eines mit der geringsten Entropie heraussucht und zurückgibt. Zuerst

wird eine neue Liste angelegt, welche die Größe Entropietoleranz+1 hat und anschließend mit leeren Listen gefüllt wird. Genauere Informationen zur Entropietoleranz befinden sich im Kapitel 3.2.5. Als Nächstes wird eine Schleife ausgeführt, welche durch alle Einträge der zweidimensionalen Liste *map* geht, welche alle Potenziale enthält. Für jeden Eintrag wird die Funktion *numberOfOnes()* (siehe 4.2) ausgeführt, um die Entropie des aktuellen Potenzials zu bestimmen. Nun wird kontrolliert, ob die Entropie höher ist als 1, was bedeutet, dass das aktuelle Potenzial noch nicht kollabiert wurde. Wenn dies der Fall ist, wird kontrolliert, ob die Entropie geringer ist als die bis jetzt gefundene geringste Entropie eines Potenzials. Falls ja, wird die aktuelle geringste Entropie aktualisiert. Als Nächstes wird die erste Dimension der vorher angelegten Liste durchlaufen. Hier werden in jedem Schritt die zuvor gefundenen Potenziale mit der gleichen Entropie, welche sich im Rahmen der Entropietoleranz befinden, um 1 nach unten verschoben. Potenziale, die vorher die geringste Entropie+1 hatten, haben nun, sofern eine neue geringste Entropie gefunden wurde, die geringste Entropie+2. So wird die Liste, welche nun außerhalb des akzeptablen Rahmens der Entropie (bzw. Entropietoleranz) liegt, verworfen und es wird eine neue Liste für die Potenziale der geringsten Entropie hinzugefügt. Das zuvor gefundene Potenzial wird nun dieser neuen Liste hinzugefügt. Für den Fall, dass dieses Potenzial nicht die geringste Entropie besitzt, wird als Nächstes kontrolliert, ob sich das Potenzial im akzeptablen Rahmen der Entropietoleranz befindet. Wenn ja, wird das neue Potenzial in der zweidimensionalen Liste der Potenziale der entsprechenden Liste der zweiten Dimension hinzugefügt.

Nachdem nun alle Potenziale durchlaufen sind, wird das Potenzial mit der höchsten Entropie herausgefunden. Als nächstes wird eine kombinierte Liste angelegt, welche die zweidimensionale Liste der Potenziale in eine eindimensionale Liste zusammenfügt. Nun folgt ein Check, für den Fall, dass kein geringstes Potenzial gefunden werden konnte. Dies kann nur der Fall sein, wenn alle Potenziale auf der Map bereits kollabiert wurden und somit kein weiteres Potenzial zum Kollabieren gefunden werden kann. Sollte dies eintreten, wird 0 zusammen mit der höchsten, gefundenen Entropie zurückgegeben. Sollte dies nicht der Fall sein, wird eine zufällige Wahl aus der kombinierten Liste zusammen mit der höchsten Entropie zurückgegeben.

Collapse Tile

In Abbildung 4.6 ist die Funktion zu sehen, welche, wenn ausgeführt, ein übergebenes Potenzial kollabiert. Zuerst wird die Anzahl der Einsen des Potenzials bestimmt, welche in einer Variablen gespeichert wird. Zusätzlich wird eine Variable *num* durch die Funktion *numberOfOnes* (siehe 4.2) angelegt. Dieser Wert beschreibt, wie viele verschiedene Zustände dieses Potenzial noch annehmen kann. Als nächstes wird eine

```

27     def findLowestEntropyTile():
28         # FIND LOWEST ENTROPY TILE
29         lowest = 10
30         highest = 1
31         listOfLowest = []
32         for i in range (0,entropyTolerance+1):
33             listOfLowest.append([])
34
35
36         for y,row in enumerate(map):
37             for x,col in enumerate(row):
38                 if (numberOfOnes(col) > 1):
39                     if (numberOfOnes(col) < lowest):
40                         lowest = numberOfOnes(col)
41                         for i in range (0,len(listOfLowest)-1):
42                             if (i != len(listOfLowest)-1):
43                                 listOfLowest[len(listOfLowest)-1-i] = \
44                                 listOfLowest[len(listOfLowest)-i-2]
45                             else:
46                                 listOfLowest[0].clear()
47
48                     listOfLowest[0].append((x,y,col))
49                 elif (numberOfOnes(col) <= lowest+entropyTolerance and
50                       numberOfOnes(col) >= lowest):
51
52                     listOfLowest[numberOfOnes(col)-lowest].append((x,y,col))

```

Abbildung 4.4.: wave.py findLowestEntropyTile() 1

```

55     for y,row in enumerate(map):
56         for x,col in enumerate(row):
57             if (numberOfOnes(col) > highest):
58                 highest = numberOfOnes(col)
59
60
61
62         listCombined = []
63         for i in listOfLowest:
64             listCombined += i
65
66         if (len(listCombined) == 0):
67             return 0, highest
68
69         return random.choice(listCombined), highest

```

Abbildung 4.5.: wave.py findLowestEntropyTile() 2

zufällige Zahl zwischen 1 und num gewählt. Diese Zahl definiert, zu welchen der verschiedenen Zustände dieses Potenzial kollabiert wird, indem sie bestimmt, die wievielte Eins im Potenzial als einzige vorhanden bleiben soll. Um nun alle anderen Zustände auf 0 zu setzen, wird eine While-Schleife benutzt, welche beendet wird, sobald die Variable r mit dem zufälligen Zustand unter 1 fällt. In jedem Durchgang wird zunächst geprüft, ob das Potenzial im binären UND , verglichen mit 2 hoch dem aktuellen Iterationsindex, ungleich 0 ist. Wenn dies der Fall ist, muss an diesem Index der Binärrepräsentation des Integers eine 1 zu finden sein. Nun wird von dem Zufallswert 1 abgezogen, weil eine Eins gefunden wurde. Sofern also der Zufallswert noch nicht 0 ist und somit der Index des zufällig ausgewählten Zustands noch nicht erreicht ist, wird die Indexvariable um eins erhöht und die Schleife fängt von vorne an. Dies wird so oft wiederholt, bis der entsprechende Index gefunden wurde. In diesem Fall wird nun die Schleife beendet, weil r kleiner ist als 1 und der finale Wert für das Potenzial wird auf 2 hoch dem aktuellen Index gesetzt. Dieser Wert wird letztlich zurückgegeben.

```

72     def collapseTile(tile):
73         t = tile
74         num = numberofOnes(t)
75         r = random.randint(1,num)
76         ind = 0
77
78         while (r>0):
79             if (t&(2**ind) != 0):
80                 r -= 1
81                 if (r == 0):
82                     t = 2**ind
83             ind+=1
84
85         return t

```

Abbildung 4.6.: wave.py collapseTile()

Combined Tile Condition

Abbildung 4.7 zeigt die Funktion, welche für ein übergebenes Potenzial die Bedingungen für benachbarte Potenziale berechnet und zurückgibt. Hierfür wird eine Variable angelegt, welche die Kombination aller Bedingungen beinhaltet. Anfänglich ist dies ein Potenzial mit allen Zuständen. Nun wird für jeden existierenden Zustand überprüft, ob dieser noch für das übergebene Potenzial möglich ist. Dies wird mit einem binären UND ermöglicht, welches das übergebene Potenzial mit der Benachbarungsbedingung des aktuellen Zustands aus der Potenzialkompatibilitätsliste vergleicht. Wenn dies nicht der Fall ist, also der Vergleich einer 0 entspricht, ist dieser aktuelle Zustand

für alle benachbarten Potenziale nicht mehr möglich. Dieses Ergebnis wird nun zu den kombinierten Bedingungen für benachbarte Potenziale hinzugefügt, indem die entsprechende Variable mit dem Ergebnis eines binären Vergleichs des vorherigen Wertes und der invertierten Identität des aktuellen Zustands überschrieben wird, welche der Potenzialkompatibilitätsliste entnommen wird. Schlussendlich kann nun dieser kombinierte Bedingungswert zurückgegeben werden.

```

87     def combinedTileCondition(tile):
88         combined = 0b11111111
89
90         for t in wfl.tileCompatibilityLookUpTable.items():
91             if (tile & t[1] == 0):
92                 combined &= t[0]^0b11111111
93
94         return combined

```

Abbildung 4.7.: wave.py combinedTileCondition()

Update Map

In Abbildung 4.8 ist die Funktion zu sehen, die für das Updaten aller Potenziale zuständig ist. Diese Funktion iteriert durch alle Potenziale in der übergebenen Liste. Im nächsten Schritt wird die Funktion *combinedTileCondition()* (siehe 4.2) aufgerufen, welche einen Binärstring mit allen Bedingungen an die benachbarten Potenziale zurückgibt. Anschließend werden diese Bedingungen auf alle benachbarten Potenziale angewendet. Sollte sich nun der Wert eines benachbarten Potenzials durch die Bedingungen geändert haben, wird dieses einer Liste hinzugefügt. Im letzten Schritt wird diese Funktion rekursiv auf alle Potenziale aus der vorher erstellten Liste angewendet. So propagiert eine Änderung an einem der Potenziale über alle benachbarten Potenziale, bis sich nach dem Updaten keine Potenziale mehr geändert haben.

Algorithm Step

Abbildung 4.9 zeigt die Funktion, welche den makroskopischen Kern des Algorithmus enthält. Zuerst werden durch die Funktion *findLowestEntropyTile()* (siehe 4.2) die Potenziale mit der höchsten und der geringsten Entropie aus der zweidimensionalen Liste an Potenzialen herausgesucht. Für den Fall, dass die höchste Entropie in der Liste 1 ist, wird *True* zurückgegeben und somit ist der Algorithmus beendet. Ist dies nicht der Fall, wird nun die Funktion *collapseTile()* (siehe 4.2) angewendet, welche das angegebene Potenzial zufällig kollabiert. Das Ergebnis dieser Funktion wird nun an derselben Position in der zweidimensionalen Liste gespeichert. Da somit eine Änderung an der Liste erfolgt ist, müssen alle benachbarten Potenziale aktualisiert werden. Dies

```

98     def updateMap(toUpdate):
99         nextUpdate = []
100        for tile in toUpdate:
101            x = tile[0]
102            y = tile[1]
103            condition = combinedTileCondition(tile[2])
104            if (x+1 < len(map[0])):
105                temp = map[y][x+1]
106                map[y][x+1] &= condition
107                if (temp != map[y][x+1]):
108                    nextUpdate.append((x+1,y,map[y][x+1]))
109            if (y+1 < len(map)):
110                temp = map[y+1][x]
111                map[y+1][x] &= condition
112                if (temp != map[y+1][x]):
113                    nextUpdate.append((x,y+1,map[y+1][x]))
114            if (x-1 >= 0):
115                temp = map[y][x-1]
116                map[y][x-1] &= condition
117                if (temp != map[y][x-1]):
118                    nextUpdate.append((x-1,y,map[y][x-1]))
119            if (y-1 >= 0):
120                temp = map[y-1][x]
121                map[y-1][x] &= condition
122                if (temp != map[y-1][x]):
123                    nextUpdate.append((x,y-1,map[y-1][x]))
124
125        if (len(nextUpdate)>0):
126            updateMap(nextUpdate)

```

Abbildung 4.8.: wave.py updateMap()

geschieht durch die Funktion *updateMap()* (siehe 4.2). Am Ende gibt die Funktion *False* zurück.

Wave Function Lookup Table

Der Wave Function Lookup Table (Abbildung 4.10) stellt eine eigene Bibliothek dar, welche drei verschiedene globale Variablen enthält. Die erste ist *tileCompatibilityList*, welche die Benachbarungsbedingungen der einzelnen Tiles enthält. Die zweite ist der *tileCompatibilityLookUpTable*, diese enthält dieselben Bedingungen in Binärrepräsentation. Die dritte ist der *BinaryLookupTable*, welcher die Binärrepräsentation für die Identität der einzelnen Tiles enthält. Diese drei Variablen werden durch eine Anfrage an den Manager-Service befüllt.

```
157 def algorithmStep():
158     lowestEntropyTile, highestEntropy = findLowestEntropyTile()
159     if (highestEntropy <= 1):
160         return True
161
162     x = lowestEntropyTile[0]
163     y = lowestEntropyTile[1]
164
165     map[y][x] = collapseTile(map[y][x])
166     updateMap([(x,y,map[y][x])])
167
168     return False
```

Abbildung 4.9.: wave.py algorithmStep()

```
1 import pandas as pd
2 import requests
3 import json
4
5
6 # WAVE FUNCTION TILE LOOKUP TABLE
7
8 # ULRs
9 managerurl = "http://wfcmanger:5000"
10
11
12 tileCompatibilityList = []
13 tileCompatibilityLookUpTable = {}
14 binaryLookUpTable = {}
15
16 def requestRestrictions():
17     global tileCompatibilityList
18     global tileCompatibilityLookUpTable
19     global binaryLookUpTable
20
21     response = requests.get(managerurl+"/restrictions").json()
22
23     temp = response[1]
24     for i in temp.items():
25         tileCompatibilityLookUpTable[int(i[0])] = i[1]
26     tileCompatibilityList = response[0]
27     binaryLookUpTable = response[2]
```

Abbildung 4.10.: Wave Function Lookup Table

Pretty Print Map

Die Funktion `prettyPrintMap()` (Abbildung 4.11) diente während der Entwicklung dazu, die Daten einer Map auszugeben. Hierbei werden keine Sprites für eine visualisierte Ausgabe verwendet, sondern es handelt sich lediglich um die Binärrepräsentationen der Tiles. Abbildung 4.12 zeigt ein Beispiel einer mit `prettyPrintMap()` ausgegebenen Map.

```

129  def prettyPrintMap(map):
130      def drawHorizontalLine(length):
131          for r in range(0,length):
132              print("-", end='')
133
134      distance = 6
135      drawHorizontalLine(distance*numberOfTiles[0]-1+11*numberOfTiles[0]+1)
136      for y in range (0,len(map[0])):
137          print("")
138          print("| ", end='')
139          for x in range (0,len(map)):
140              print(bin(map[y][x]), end=' ')
141              fill = 11 - len(bin(map[y][x]))
142              for f in range (0,fill):
143                  print(" ", end='')
144                  for a in range (0,int(distance/2)):
145                      print(" ", end='')
146                      print("| ", end='')
147                      for a in range (0,int(distance/2)):
148                          print(" ", end='')
149              print("")
150              drawHorizontalLine(distance*numberOfTiles[0]-1+11*numberOfTiles[0]+1)
151      print("")
```

Abbildung 4.11.: wave.py prettyPrintMap()

0b10000000 0b10000000 0b10000000 0b10000000
0b10000000 0b10000000 0b10000000 0b10000000
0b10000000 0b10000000 0b10000000 0b10000000
0b111000000 0b111000000 0b10000000 0b10000000

Abbildung 4.12.: Beispiel einer Ausgabe mit `prettyPrintMap()`

4.3. Services

4.3.1. Manager

Der Manager dient als zentrale Anwendung für die Festlegung der Regeln (*Rules* siehe Kapitel 3.2.3) und Bedingungen (*Restrictions* siehe Kapitel 3.2.4), auf deren Basis eine Map erstellt wird. Andere Dienste können via POST-Call Anfragen an den Manager senden, um die Werte für die verschiedenen Rules und Restrictions zu erhalten. Die Rules werden beim Start aus einer Tabelle ausgelesen und beinhalten folgende Werte:

- **numberOfTiles**: die Größe der zu erstellenden Map in Anzahl der Tiles als Tupel (x- und y-Achse)
- **numberOfParts**: die Anzahl der Map Parts (*Chunks*), in welche die gesamte Map zerlegt werden soll. Der Wert unterliegt speziellen Bedingungen (siehe Kapitel 3.3)
- **entropyTolerance**: Der Wert der Entropietoleranz beim Erstellen der Map (siehe auch Abschnitt 3.2.5)
- **numberOfWorkers**: die Anzahl der zu verwendenden Worker

Die Restrictions werden aus einer eigenen Tabelle ausgelesen und beinhalten die Bedingungen der Tiles zum Aufbau der Map (siehe auch Abschnitt Restrictions 3.2.4).

Dockerfile

Das Image des Manager-Services wird mithilfe einer Dockerfile erstellt (Abbildung 4.13). Auf Basis eines Alpine-Betriebssystems werden zunächst alle benötigten Bibliotheken installiert, bevor das Manager-Python-Skript sowie das HTML-Template für die Homepage in das Image kopiert werden. Zuletzt wird der Einstiegspunkt konfiguriert, in dem die Python Flask-Applikation mit den entsprechenden Attributen des Managers und dem Port 5000 gestartet wird, welcher zur Kontaktaufnahme anderer Services dient.

```

1  FROM python:alpine3.19
2  RUN pip install pandas flask openpyxl
3  ADD ..
4  EXPOSE 5000
5  CMD ["python", "-m", "flask", "--app", "manager", "run", "--host=0.0.0.0",
6    |   "--port", "5000"]

```

Abbildung 4.13.: Manager Dockerfile

Template

Die Homepage des Manager-Services basiert auf einem einfachen HTML-Template (Abbildung 4.14) und bietet Eingabefelder für die Werte *numberOfTiles*, *numberOfParts*, *entropyTolerance* und *numberOfWorkers*. Dazu verfügt die Seite über einen Button, mit dem die eingegebenen Werte als Set von *Rules* gespeichert werden können. So ist es möglich, im laufenden Betrieb die Werte zu verändern, bevor die Generierung einer Map gestartet wird (Abbildung 4.15).

```

1  <!DOCTYPE html>
2  <html lang="de">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Manager Service</title>
7  </head>
8  <body>
9      <h1>MANAGER SERVICE</h1>
10     <form method="POST">
11         <label for="var1">numberOfTiles:</label>
12         <input type="text" id="var1" name="var1" value="{{ var1 }}"/><br><br>
13
14         <label for="var2">numberOfParts:</label>
15         <input type="text" id="var2" name="var2" value="{{ var2 }}"/><br><br>
16
17         <label for="var3">entropyTolerance:</label>
18         <input type="text" id="var3" name="var3" value="{{ var3 }}"/><br><br>
19
20         <label for="var4">numberOfWorkers:</label>
21         <input type="text" id="var4" name="var4" value="{{ var4 }}"/><br><br>
22
23         <button type="submit">set rules</button>
24     </form>
25 </body>
26 </html>

```

Abbildung 4.14.: Manager Template

MANAGER SERVICE

numberOfRowsTiles:
 numberOfRowsParts:
 entropyTolerance:
 numberOfWorkers:

Abbildung 4.15.: Manager Homepage

Imports und Rules/Restrictions

Nach dem Start liest der Dienst die Dateien *rules.xlsx* und *restrictions.xlsx* aus, erstellt mithilfe der Werte eine Liste für Rules und Tile-Kompatibilität und erzeugt die Tabellen *tileCompatibilityLookUpTable* und *binaryLookUpTable* (Abbildung 4.16 und 4.17).

```

1  import pandas as pd
2  from flask import Flask, request, render_template, redirect, jsonify
3
4
5  # MANAGER
6  app = Flask(__name__)
7
8  # READ RULES
9  data = pd.read_excel("rules.xlsx", usecols="B")
10
11 app.config["numberOfTiles"] = int(data.values[0][0])
12 app.config["numberOfParts"] = int(data.values[2][0])
13 app.config["entropyTolerance"] = int(data.values[3][0])
14 app.config["numberOfWorkers"] = int(data.values[4][0])
15
16 print("numberOfTiles: ", app.config["numberOfTiles"])
17 print("numberOfParts: ", app.config["numberOfParts"])
18 print("entropyTolerance: ", app.config["entropyTolerance"])
19 print("numberOfWorkers: ", app.config["numberOfWorkers"])

```

Abbildung 4.16.: Manager Imports und Read Rules

```

76 # WAVE FUNCTION TILE LOOKUP TABLE
77 data = pd.read_excel("restrictions.xlsx", usecols="AI")
78
79 tileCompatibilityList = []
80 tileCompatibilityLookUpTable = {}
81
82 ind = 0
83
84 for tile in data.values:
85     tileCompatibilityList.append(int(tile[0],2))
86     tileCompatibilityLookUpTable[2**ind] = int(tile[0],2)
87     ind += 1
88
89
90 binaryLookUpTable = {"grass":0b00000001,
91                      "wald":0b00000010,
92                      "kuh":0b000000100,
93                      "strand":0b000001000,
94                      "wasser":0b000010000,
95                      "fisch":0b000100000,
96                      "berg":0b001000000,
97                      "bergschnee":0b010000000,
98                      "schneemann":0b100000000}

```

Abbildung 4.17.: Manager Tile Lookup

Routen

Nachdem die benötigten Tabellen erzeugt wurden, bietet der Dienst folgende Routen an (Abbildung 4.18 und 4.19):

- / (**homepage**): Zeigt die aktuell vorhandenen *Rules* im Browser an.
- /**numberOfTiles**: Antwortet mit dem Wert der Variablen *numberOfTiles*.
- /**numberOfParts**: Antwortet mit dem Wert der Variablen *numberOfParts*
- /**numberOfWorkers**: Antwortet mit dem Wert der Variablen *numberOfWorkers*
- /**entropyTolerance**: Antwortet mit dem Wert der Variablen *entropyTolerance*.
- /**restrictions**: Antwortet mit einer Liste aus *tileCompatibilityList*, *tileCompatibilityLookUpTable* und *binaryLookUpTable*.

```
22 # MANAGER SERVICE PATHS
23
24 @app.route("/")
25 def showRules():
26     return Flask.jsonify({"numberOfTiles": ": (app.config["numberOfTiles"],
27 |                                         app.config["numberOfTiles"]),
28 |                                         "numberOfParts": ": app.config["numberOfParts"],
29 |                                         "entropyTolerance": ": app.config["entropyTolerance"],
30 |                                         "numberOfWorkers": ": app.config["numberOfWorkers"]}")
31
32 @app.route('/setRules', methods=['GET', 'POST'])
33 def showHome():
34     if request.method == 'POST':
35         # processing current values
36         app.config["numberOfTiles"] = int(request.form.get('var1'))
37         app.config["numberOfParts"] = int(request.form.get('var2'))
38         app.config["entropyTolerance"] = int(request.form.get('var3'))
39         app.config["numberOfWorkers"] = int(request.form.get('var4'))
40         # save or process values
41         return render_template('manager.html',
42                               var1=app.config["numberOfTiles"],
43                               var2=app.config["numberOfParts"],
44                               var3=app.config["entropyTolerance"],
45                               var4=app.config["numberOfWorkers"])
46     return render_template('manager.html',
47                           var1=app.config["numberOfTiles"],
48                           var2=app.config["numberOfParts"],
49                           var3=app.config["entropyTolerance"],
50                           var4=app.config["numberOfWorkers"])
```

Abbildung 4.18.: Manager Routes 1

```
53     @app.route("/numberOfTiles")
54     def getNumberOfTiles():
55         return jsonify((app.config["numberOfTiles"],app.config["numberOfTiles"]))
56
57     @app.route("/numberOfParts")
58     def getNumberOfParts():
59         return jsonify(app.config["numberOfParts"])
60
61     @app.route("/entropyTolerance")
62     def getEntropyTolerance():
63         return jsonify(app.config["entropyTolerance"])
64
65     @app.route("/numberOfWorkers")
66     def getNumberOfWorkers():
67         return jsonify(app.config["numberOfWorkers"])
68
69     @app.route("/restrictions")
70     def getRestrictions():
71         return jsonify((tileCompatibilityList,tileCompatibilityLookUpTable,
72                         | | | | binaryLookUpTable))
```

Abbildung 4.19.: Manager Routes 2

4.3.2. Distributor

Der Distributor dient zur Vorberechnung und Verteilung der einzelnen Map-Abschnitte (*Chunks*). Er verfügt über ein Frontend, in dem der Generierungsprozess einer Map gestartet werden kann. Während des Prozesses wird zunächst eine Anfrage an den Manager-Service geschickt, um die entsprechenden Werte für *Rules* und *Restrictions* zu erhalten. Mit Hilfe dieser Daten wird dann eine Map erstellt, welche anschließend in die gegebene Anzahl an Chunks geteilt wird (siehe auch Kapitel 3.3). Für jeden dieser Map-Chunks werden die Ränder berechnet und mit Tiles gefüllt. Anschließend wird jeder Chunk mit einer eigenen Chunk-ID versehen und zusammen mit der Map-ID an den Hub-Service gesendet. Nachdem der letzte Chunk erstellt wurde, ist der Generierungsprozess beendet. Die Map-ID der zu generierenden Map wird auf dem Frontend des Services ausgegeben (siehe Abbildung 4.24).

Dockerfile

Das Image des Distributor-Services wird durch eine Dockerfile erstellt (Abbildung 4.20). Das Image basiert auf einem Alpine-Betriebssystem, auf dem zunächst die benötigten Bibliotheken installiert werden. Anschließend wird das Template für das Frontend sowie das Python-Skript in das Image kopiert. Zuletzt wird der Einstiegpunkt für den Dienst konfiguriert.

```
1  FROM python:alpine3.19
2  RUN pip install requests flask openpyxl pandas
3  ADD . .
4  EXPOSE 5001
5  CMD ["python","-m","flask","--app","distributor","run","--host=0.0.0.0",
6    |   |   "--port","5001"]
```

Abbildung 4.20.: Distributor Dockerfile

Imports und Routen

Nachdem der Dienst gestartet wurde, werden zunächst alle benötigten Pakete importiert (Abbildung 4.21). Danach bietet der Dienst folgende Routen an (Abbildung 4.22):

- / (homepage): Die Homepage des Distributor Services

- **/mapGenerator:** Seite für die Generierung einer Map. Nachdem ein Generierungsprozess abgeschlossen wurde, wird die Map-ID hier ausgegeben (siehe Abbildung 4.24).

```
1  import wave
2  import math
3  import requests
4  from flask import Flask, request, render_template, redirect
5  import uuid
6  import json
7  import time
8  from datetime import datetime
```

Abbildung 4.21.: Distributor Imports

```
11 app = Flask(__name__)
12
13 # URLs
14 huburl = "http://wfchub:5002"
15 managerurl = "http://wfcmanager:5000"
16 timekeeperurl = "http://wfctimekeeper:6002"
17
18 # DISTRIBUTOR SERVICE PATHS
19 @app.route("/")
20 def showHome():
21     return "SERVICE FOR DISTRIBUTING MAPS"
22
23 @app.route("/mapGenerator",methods=["GET","POST"])
24 def mapGenerator():
25     if request.method == 'POST':
26         mapuuid = generateMap()
27         return render_template('distributor.html')+"\n<H1>"+ mapuuid +"</H1>"
28     elif request.method == 'GET':
29         return render_template('distributor.html')
```

Abbildung 4.22.: Distributor Routes

Template

Die Homepage des Distributor-Services basiert auf einem einfachen HTML-Template (Abbildung 4.23) und beinhaltet einen Button, um den Generierungsprozess einer Map zu starten.

```

1  <html>
2  <body>
3  <font face="Arial"></font>
4  <p>MAP GENERATOR</p>
5  <form method="post" action="/mapGenerator">
6  <input type="submit" name="generate_button" value="generate map">
7  </form>
8  </body>
9  </html>
```

Abbildung 4.23.: Distributor Template

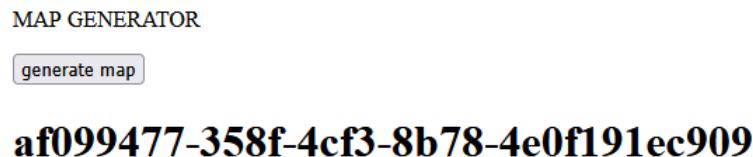


Abbildung 4.24.: Distributor Homepage

Generate Map

Sobald ein Generierungsprozess gestartet wurde, wird die Funktion *gererateMap()* ausgeführt (Abbildung 4.25-4.27) und das Programm beginnt damit, sich die *Rules* und *Restrictions* durch einen Request an den Manager-Service einzuholen. Dabei werden die Werte *numberOfTiles*, *numberOfParts*, *entropyTolerance* und *numberOfWorkers* übertragen (Abbildung 4.28). Sollte die Verbindung zum Manager fehlschlagen, wird nach einer Pause von 30 Sekunden ein erneuter Verbindungsversuch durchgeführt. Sobald alle Daten erhalten wurden, wird mit *setMap()* (siehe 4.3.2) zunächst eine leere Map in der vorgegebenen Größe erstellt. Anschließend wird die Funktion *distrubuteMap()* (siehe 4.3.2) aufgerufen, um die einzelnen Chunks zu erstellen. Sobald die Chunks erstellt wurden, werden sie zusammen mit ihren jeweiligen Attributen sowie der zugehörigen Map-ID an den Hub-Service übertragen (Abbildung 4.26). Anschließend wird ein Zeitstempel gesetzt, um für spätere Messungen den Startzeitpunkt des Generierungsprozesses festzuhalten. Dieser wird zusammen mit den Attributen sowie der ID der Map an den Timekeeper-Dienst gesendet (Abbildung 4.27). Danach ist der Generierungsvorgang seitens des Distributors abgeschlossen und die Map-ID wird auf der Homepage des Dienstes ausgegeben (Abbildung 4.24).

```

102     def generateMap():
103         # connects to manager service to get the current rule set.
104         # creates and distributes a map and sends it to huib service.
105         while True:
106             try:
107                 print("getting rules")
108                 rules = getRules()
109                 print("rules: ", rules)
110                 print("")
111                 break
112             except:
113                 print("Connection Failed")
114                 time.sleep(30)
115             print("connection success")
116             wave.numberOfTiles = rules["numberOfTiles"]
117             wave.entropyTolerance = rules["entropyTolerance"]
118             fullMap = setMap(rules["numberOfTiles"])
119             wave.map = fullMap
120             mapChunks = distributeMap(fullMap, rules["numberOfParts"])

```

Abbildung 4.25.: Distributor generateMap() 1

```

122     # SEND PARTS TO HUB
123     data = []
124     mapID = str(uuid.uuid4()) # generate mapID
125     print("sending chunks to hub..")
126     for x in range(0,len(mapChunks[0])):
127         for y in range(0,len(mapChunks)):
128             data.append({"mapID":mapID,"chunkID":str(uuid.uuid4()),
129                         "locX":x,"locY":y,
130                         "entropyTolerance":rules["entropyTolerance"],
131                         "content":mapChunks[y][x]})
132     obj = json.dumps(data)
133     result = requests.post(huburl+"/saveChunks", json=obj)
134     print("done.")
135     print("")

```

Abbildung 4.26.: Distributor generateMap() 2

Get Rules

In Abbildung 4.28 ist die Funktion zu sehen, welche dafür zuständig ist, mit dem Manager-Service zu interagieren und die benötigten Parameter anzufordern. Diese sind *numberOfTiles*, *numberOfParts*, *entropyTolerance* und *numberOfWorkers*.

```

137     #SEND DATA TO TIME KEEPER
138     print("getting startTime..")
139     startTime = datetime.now()
140     startTime = startTime.isoformat()
141     print("getting map info..")
142     tdata = {"mapID":mapID,"mapSize":rules["numberOfTiles"][0],
143               "chunkCount":rules["numberOfParts"],
144               "numberOfWorkers":rules["numberOfWorkers"],
145               "startTime":startTime,"endTime":None,"totalDuration":None}
146     tobj = json.dumps(tdata)
147     print("sending data to timekeeper..")
148     result = requests.post(timekeeperurl+"/saveMapTime", json=tobj)
149     print("done..")
150     print("")
151     return mapID

```

Abbildung 4.27.: Distributor generateMap() 3

```

32     def getRules():
33         # request rules from manager service
34         numberoftilesResponse = requests.get(managerurl+"/numberOfTiles").json()
35         numberoftiles = (numberoftilesResponse[0],numberoftilesResponse[1])
36         print("numberOfTiles: ", numberoftiles)
37         numberofpartsResponse = requests.get(managerurl+"/numberOfParts").json()
38         numberofparts = numberofpartsResponse
39         print("numberOfParts: ", numberofparts)
40         entropytoleranceResponse = requests.get(managerurl+"/entropyTolerance").json()
41         entropytolerance = entropytoleranceResponse
42         print("entropyTolerance: ", entropytolerance)
43         numberofworkersResponse = requests.get(managerurl+"/numberOfWorkers").json()
44         numberofworkers = numberofworkersResponse
45         print("numberOfWorkers: ", numberofworkers)
46
47         return {"numberOfTiles":numberoftiles, "numberOfParts":numberofparts,
48                 "entropyTolerance":entropytolerance, "numberOfWorkers":numberofworkers}

```

Abbildung 4.28.: Distributor getRules()

Set Map

Abbildung 4.29 zeigt die Funktion, welche bei Aufruf eine leere Map erzeugt, indem sie eine zweidimensionale Liste anlegt. Die gewollte Größe wird der Funktion übergeben. Die Funktion füllt die neu erstellte Map im selben Schritt durch Python's List-Comprehension mit dem Integer 511, dessen Binärrepräsentation durch '11111111' dargestellt wird (siehe auch Kapitel 4.1.1). Letztlich wird die neue Map zurückgegeben.

```
51  def setMap(t):
52      # sets an empty map with t*t size
53      fullMap = [[0b1111111111 for x in range(0,t[0])] for y in range(0,t[1])]
54      return fullMap
```

Abbildung 4.29.: Distributor setMap()

Distribute Map

Die Funktion *distributeMap()* dient dazu, die vorliegende Map in die angegebene Anzahl von Chunks zu unterteilen und mithilfe des WFC-Algorithmus die Ränder der einzelnen Abschnitte zu berechnen (Abbildung 4.30, 4.31).

Zuerst wird die Anzahl der benötigten Divisionen in der X- und Y-Dimension (Länge, Breite) berechnet, welche die Quadratwurzel aus der Gesamtanzahl der zu verteilenden Teile ist. Das Format der kleineren Teile der Map spiegelt dabei das Format der gesamten Map wider, dementsprechend sollen diese auch zweidimensionale Listen mit Integer-Werten sein. Um dies zu erreichen, wird eine vierfach verschachtelte For-Schleife benötigt. Diese iteriert über die zwei Dimensionen der originalen Map in Schritten gleich der Anzahl der benötigten Divisionen. Mit den zwei inneren Schleifen wird nun die benötigte Datenstruktur angelegt. Hier folgt nun ein Test, ob die Datenstruktur so verwendet werden kann, oder die Anzahl der Divisionen nicht möglich ist. Im nächsten Schritt wird die Map nun an der „Naht“ entlang auf beiden Seiten berechnet. Hierfür wird auf sowohl der x- als auch der y-Achse über jede Division iteriert. Für jede Division werden die Tiles dem Rand entlang kollabiert und die Map upgedatet (siehe Abschnitt 4.2). Die benachbarten Tiles werden durch dieses Update mitbeeinflusst, dementsprechend wird bei der Verteilung mehr als nur der Rand der Naht berechnet. So entstehen mit Tiles vorberandete Abschnitte der Map (siehe Abbildung 4.32 und 4.33). Im letzten Schritt wird die oben angelegte Datenstruktur mit der nun teilweise vorberechneten und unterteilten Map gefüllt und die Liste mit Map-Chunks wird zurückgegeben.

```

58     def distributeMap(map, numberOfParts):
59         # divides map in numberOfParts (chunks), calculates borders
60         divisions = int(math.sqrt(numberOfParts))
61         mapChunks = []
62         for i in range (0,divisions):
63             mapChunks.append([])
64             for j in range (0,divisions):
65                 mapChunks[i].append([])
66                 for k in range (0,int(len(map)/divisions)):
67                     mapChunks[i][j].append([])
68                     for l in range (0,int(len(map[0])/divisions)):
69                         mapChunks[i][j][k].append(0)
70
71
72         if (len(map) % divisions != 0) or (len(map[0]) % divisions != 0):
73             print("MAP SIZE NOT DISTRIBUTABLE")
74             return 0

```

Abbildung 4.30.: Distributor distributeMap() 1

```

76     for y in range (0,len(map)):
77         for i in range (1, int(divisions)):
78             x = int(len(map[0])/(divisions))*i-1
79             map[y][x] = wave.collapseTile(map[y][x])
80             wave.updateMap([(x,y,map[y][x])])
81             map[y][x+1] = wave.collapseTile(map[y][x+1])
82             wave.updateMap([(x+1,y,map[y][x+1])])
83         for x in range (0,len(map[0])):
84             for i in range (1, int(divisions)):
85                 y = int(len(map)/(divisions))*i-1
86                 map[y][x] = wave.collapseTile(map[y][x])
87                 wave.updateMap([(x,y,map[y][x])])
88                 map[y+1][x] = wave.collapseTile(map[y+1][x])
89                 wave.updateMap([(x,y+1,map[y+1][x])])
90
91
92         for x in range (0,len(map[0])):
93             for y in range (0,len(map)):
94                 mapChunks[int(y/(len(map)/divisions))]\ \
95                 [int(x/(len(map[0])/divisions))]\ \
96                 [y%int(len(map)/divisions)][x%int(len(map[0])/divisions)]\ \
97                 = map[y][x]
98     return mapChunks

```

Abbildung 4.31.: Distributor distributeMap() 2

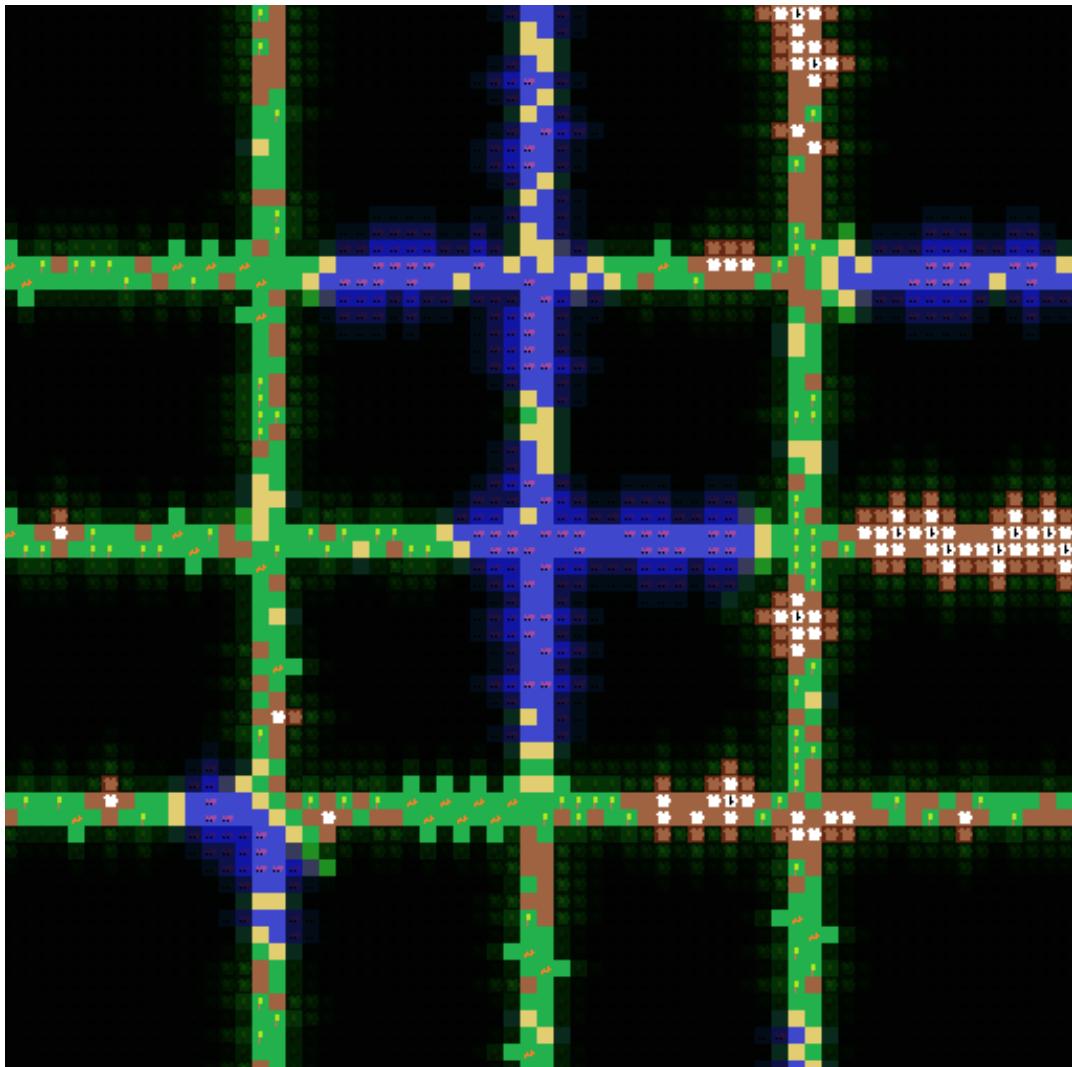


Abbildung 4.32.: Beispiel einer vorberechneten Map mit 16 Parts

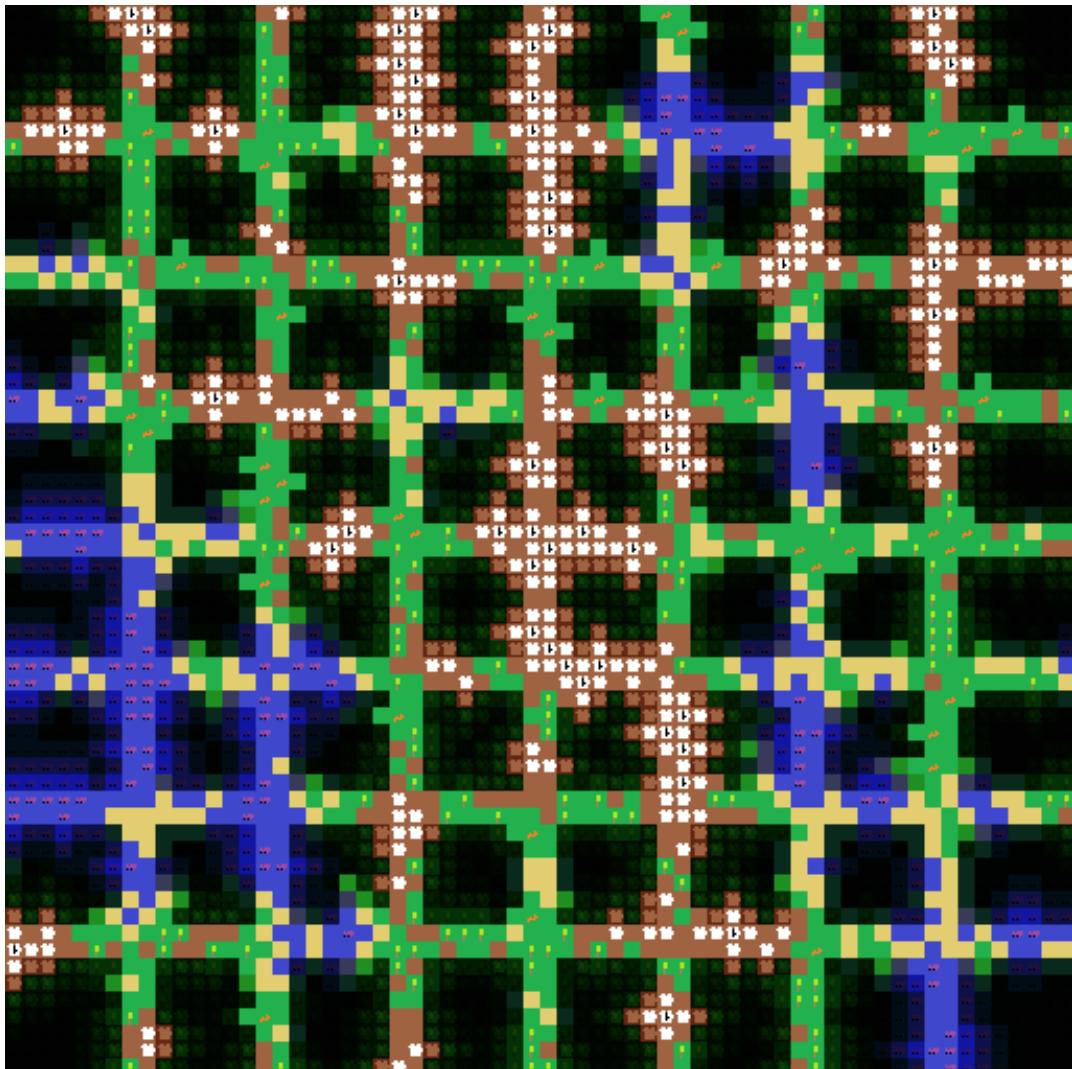


Abbildung 4.33.: Beispiel einer vorberechneten Map mit 64 Parts

4.3.3. Hub

Der Hub dient zur Verwaltung von Chunks und verfügt innerhalb des Clusters über eine alleinige Verbindung zur Hub-Datenbank.

Dockerfile

Das Image des Hub-Services wird durch eine Dockerfile erstellt (Abbildung 4.34). Es basiert auf einer für Python optimierten Version eines Alpine-Betriebssystems, auf dem zunächst alle benötigten Bibliotheken installiert werden. Anschließend wird das Python-Skript in das Image kopiert. Im letzten Schritt wird der Einstiegspunkt des Programms konfiguriert.

```
1  FROM python:alpine3.19
2  RUN pip install mysql-connector-python flask pika
3  ADD ..
4  EXPOSE 5002
5  CMD ["python", "-m", "flask", "--app", "hub", "run", "--host=0.0.0.0",
6    |   |   | "--port", "5002"]
```

Abbildung 4.34.: Hub Dockerfile

Imports und globale Variablen

Nachdem der Dienst gestartet wurde, werden zunächst alle benötigten Pakete importiert (Abbildung 4.35). Zusätzlich werden verschiedene globale Variablen festgelegt, wie die Adressen der Datenbank und des RabbitMQ-Dienstes sowie verschiedener vorgefertigter SQL-Befehle zum Speichern und Aktualisieren von Chunks in der Datenbank (Abbildung 4.36).

```
1  import mysql.connector
2  import uuid
3  import json
4  from flask import Flask, request
5  import pika
```

Abbildung 4.35.: hub.py Imports

Routen

Der Dienst kann von außen über folgende Routen angesprochen werden:

```

10 # GLOBAL VARIABLES
11 dbhost = "wfcdb"
12 rabbithost = "wfcrabbit"
13 sqlInsert = "INSERT INTO mapchunks (mapID, chunkID, locationX, locationY, \
14 | | | | entropyTolerance, content, computed) " \
15 | | | | "VALUES (%s, %s, %s, %s, %s, %s, %s);"
16 sqlGetByTicketID = "SELECT * FROM mapchunks WHERE chunkID = %s;"
17 sqlUpdateChunk = "UPDATE mapchunks SET content = %s, computed = 1 " \
18 | | | | "WHERE chunkID = %s;"
19 sqlMapByID = "SELECT locationX,locationY,content " \
20 | | | | "FROM mapchunks WHERE mapID = %s;"
```

Abbildung 4.36.: hub.py Global Variables

- **/saveChunk:** ein Map-Chunk wird in der Datenbank abgelegt
- **/saveChunks:** Eine Reihe von Chunks wird in der Datenbank abgelegt
- **/updateChunkByID:** bereits bestehende Chunks können anhand ihrer ID aktualisiert werden
- **/getMapChunkByChunkID:** Der Dienst liefert den gewählten Chunk anhand der übergebenen ID
- **/getMapByID:** Der Dienst liefert die gewählte Map anhand der übergebenen ID

Die Route */saveChunk* öffnet eine Verbindung zur Datenbank und speichert unter Verwendung des vorgefertigten SQL-Befehls die Daten des übermittelten Chunks in der Tabelle *mapChunks* (Abbildung 4.37). Der Datensatz enthält neben der Map-ID auch die jeweilige Chunk-ID sowie die Information, an welcher Stelle der Map sich der Chunk befindet (*locX, locY*). Dazu kommen neben dem Wert der Entropietoleranz und dem konkreten Inhalt (*content*) des Chunks, welcher vorerst leer ist, auch die Boolean-Variable *computed*, welche aussagt, ob der Chunk bereits vollständig berechnet wurde. Beim Erstellen des Chunks wird *computed* zunächst auf *False* gesetzt.

Die Route */updateChunkByID* öffnet ebenfalls eine Verbindung zur Datenbank. Allerdings wird hier nun der Inhalt eines bereits vorhandenen Chunks aktualisiert. Anhand der Chunk-ID können die empfangenen Daten eindeutig zugeordnet - und der Inhalt des entsprechenden Chunks gefüllt werden (Abbildung 4.38).

Die Route */saveChunks* dient dazu, eine Reihe bzw. ein komplettes Set an Chunks aufzunehmen und in der Datenbank zu speichern. Neben der Verbindung zur Datenbank wird hier auch eine Verbindung zum RabbitMQ-Service geöffnet, welcher die Message-Queue *maptickets* bereitstellt. Die Daten der übermittelten Chunks werden mittels vorgefertigtem SQL-Befehl in der Datenbank abgelegt. Für jeden erhaltenen und

```
26 # HUB SERVICE
27 app = Flask(__name__)
28
29 # HUB SERVICE PATHS
30 @app.route("/")
31 def showHome():
32     return "HUB SERVICE FOR SAVING MAPCHUNKS"
33
34 @app.route("/saveChunk", methods=["POST"])
35 def saveChunk():
36     database = mysql.connector.connect(host=dbhost, database="maps",
37                                         user="root", password="root")
38     dbCursor = database.cursor()
39     data = json.loads(request.json)
40     valuesToInsert = (data["mapID"], data["chunkID"],
41                       data["locX"], data["locY"],
42                       data["entropyTolerance"], data["content"], False)
43     dbCursor.execute(sqlInsert, valuesToInsert)
44     database.commit()
45     print("saved chunk in db")
46     database.disconnect()
47     return "done"
```

Abbildung 4.37.: hub.py Route /saveChunk

```
49 @app.route("/updateChunkByID", methods=["POST"])
50 def updateChunk():
51     database = mysql.connector.connect(host=dbhost, database="maps",
52                                         user="root", password="root")
53     dbCursor = database.cursor()
54     data = json.loads(request.json)
55     valuesToInsert = (json.dumps(data["content"]), data["chunkID"])
56     dbCursor.execute(sqlUpdateChunk, valuesToInsert)
57     database.commit()
58     print("updated chunk in db")
59     database.disconnect()
60     return "done"
```

Abbildung 4.38.: hub.py Route /updateChunkByID

gespeicherten Chunk wird die jeweilige Chunk-ID an die Message-Queue des RabbitMQ-Services gesendet (Abbildung 4.39).

Die Route `/getMapChunkByChunkID` erfordert die Übermittlung einer Chunk-ID in Form einer UUID an den Hub-Service und dient dazu, die Daten eines bestimmten Chunks beim Hub-Service anzufordern. Durch die übergebene Chunk-ID können die benötigten Informationen gezielt aus der Datenbank geladen und zurückgegeben werden.

```

63     @app.route("/saveChunks", methods=["POST"])
64     def saveChunks():
65         # RABBIT CONNECTION
66         connection = pika.BlockingConnection(
67             pika.ConnectionParameters(host=rabbithost))
68         channel = connection.channel()
69         channel.queue_declare(queue='maptickets', durable=True)
70         # DB CONNECTION
71         database = mysql.connector.connect(host=dbhost, database="maps",
72             user="root", password="root")
73         dbCursor = database.cursor()
74         data = json.loads(request.json)
75         valuesToInsert = []
76         for chunk in data:
77             valuesToInsert.append((chunk["mapID"], chunk["chunkID"],
78                 chunk["locX"], chunk["locY"],
79                 chunk["entropyTolerance"],
80                 json.dumps(chunk["content"]),
81                 False))
82             message = chunk["chunkID"]
83             channel.basic_publish(
84                 exchange='', routing_key='maptickets', body=message,
85                 properties=pika.BasicProperties(delivery_mode=
86                     pika.DeliveryMode.Persistent))
87             dbCursor.executemany(sqlInsert, valuesToInsert)
88             database.commit()
89             database.disconnect()
90             channel.close() # close rabbit channel
91             connection.close() # close rabbit connection
92             print(dbCursor.rowcount)
93             print("saved set of chunks in db")
94             return "done"

```

Abbildung 4.39.: hub.py Route /saveChunks

werden (Abbildung 4.40).

Die Route */getMapByMapID* erfordert die Übermittlung einer Map-ID in Form einer UUID und dient dazu, die Daten einer gesamten Map aus der Datenbank auszulesen. Der Hub-Service öffnet eine Verbindung zur Datenbank und liest dort alle zu der übermittelten Map-ID gehörenden Chunks aus. Diese werden noch in der Funktion zusammengesetzt und als vollständige Map zurückgegeben (Abbildung 4.41).

```

96     @app.route("/getMapChunkByChunkID<uuid:ID>", methods=["GET"])
97     def getMapChunkByChunkID(ID):
98         database = mysql.connector.connect(host=dbhost, database="maps",
99                                              user="root", password="root")
100        dbCursor = database.cursor()
101        dbCursor.execute(sqlGetByTicketID, (str(ID),))
102        result = dbCursor.fetchone()
103        database.disconnect()
104        return json.dumps(result)

```

Abbildung 4.40.: hub.py Route /getMapChunkByChunkID

```

106    @app.route("/getMapByID<uuid:ID>", methods=["GET"])
107    def getMapByID(ID):
108        database = mysql.connector.connect(host=dbhost, database="maps",
109                                              user="root", password="root")
110        dbCursor = database.cursor()
111        dbCursor.execute(sqlMapByID, (str(ID),))
112        result = dbCursor.fetchall()
113        content = []
114        for chunk in result:
115            content.append(json.loads(chunk[2]))
116        sizeX = len(content[0][0])
117        sizeY = len(content[0])
118        print("sizeX: ", sizeX)
119        print("sizeY: ", sizeY)
120
121        maxLocX = -1
122        maxLocY = -1
123        for chunk in result:
124            if (chunk[0] > maxLocX):
125                maxLocX = chunk[0]
126            if (chunk[1] > maxLocY):
127                maxLocY = chunk[1]
128        fullMap = []
129        for y in range(0,(maxLocY+1)*sizeY):
130            temp = []
131            for x in range(0,(maxLocX+1)*sizeX):
132                temp.append(0)
133            fullMap.append(temp)
134        for i,chunk in enumerate(result):
135            for y in range(0,sizeY):
136                for x in range(0, sizeX):
137                    fullMap[chunk[1]*sizeY+y][chunk[0]*sizeX+x] = content[i][y][x]
138    return fullMap

```

Abbildung 4.41.: hub.py Route /getMapByMapID

4.3.4. Hub-DB

In der Hub-Datenbank werden alle eingehenden Chunks mit den folgenden Attributen abgelegt:

- **mapID**: Die ID der Map zu welcher der zu speichernde Chunk gehört
- **chunkID**: Die ID des zu speichernden Chunks
- **locX**: die X-Position des Chunks in der Map
- **locY**: die Y-Position des Chunks in der Map
- **entropyTolerance**: Die Entropietoleranz der Map
- **content**: der Inhalt des Chunks, bestehend aus einer Tabelle mit Integer-Werten.
Beim Anlegen eines Chunks in der Datenbank ist der Content zunächst leer und kann im weiteren Verlauf aktualisiert werden.
- **computed**: Boolean-Variable welche angibt, ob der Inhalt des Chunks bereits berechnet wurde. Sind die *computed*-Werte aller Chunks zu einer bestimmten Map-ID *True* gilt die Map als vollständig berechnet.

Dockerfile und Init

Das Image des Hub-DB-Services wird durch eine Dockerfile erstellt (Abbildung 4.42). Das Image basiert auf einem von Oracle zur Verfügung gestellten MySQL-Image, auf dem die vorgefertigte Init-Datei kopiert wird. Die Init-Datei erzeugt bei Start des Dienstes die Datenbank und wählt diese aus. Anschließend wird eine Tabelle gemäß den oben angegebenen Attributen der Maps bzw. Chunks angelegt (Abbildung 4.43). Der Hub-Service verfügt innerhalb des Clusters über eine alleinige Verbindung zur Hub-DB. Alle Vorgänge zum Speichern von Chunks sowie alle Anfragen zum Auslesen von Map-Daten laufen demnach immer über den Hub-Service.

```
1  FROM mysql
2  ADD ./init.sql /docker-entrypoint-initdb.d/
```

Abbildung 4.42.: Hub-DB Dockerfile

```
1 CREATE DATABASE IF NOT EXISTS maps;
2 USE maps;
3 CREATE TABLE IF NOT EXISTS mapchunks (
4     mapID VARCHAR(36),
5     chunkID VARCHAR(36),
6     locationX INT,
7     locationY INT,
8     entropyTolerance INT,
9     content JSON,
10    computed BOOLEAN,
11    PRIMARY KEY (mapID, chunkID)
12 );
```

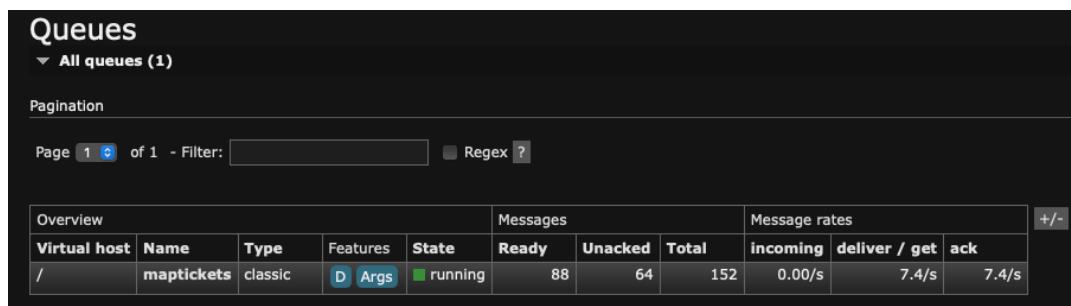
Abbildung 4.43.: Hub-DB Init

4.3.5. RabbitMQ

Der RabbitMQ-Service stellt die Message-Queue für diese Architektur zur Verfügung. Der Dienst bildet in der Architektur keine eigene Anwendung und wird ausschließlich durch das Kubernetes-Deployment (siehe Abschnitt Deployment 4.5.2) gestartet und konfiguriert. Die ID jedes beim Hub-Service eingehenden Chunks wird vom Hub als *ticket* an den RabbitMQ gesendet und dort in dem Channel *maptickets* abgelegt. Jeder Worker horcht auf diesen Channel (siehe Abschnitt 4.3.6) und holt sich die entsprechende ID aus der Queue. Bei mehreren Workern werden die IDs im Round-Robin-Verfahren verteilt.

Rabbit Management-Plugin

Das Management-Plugin ist eine in RabbitMQ integrierte Funktion und bietet dem Benutzer verschiedene Möglichkeiten des Überwachens (Monitoring). Sobald der Service gestartet ist, kann das Management über die Adresse [http://\[MANAGER-IP\]:31672](http://[MANAGER-IP]:31672) aufgerufen werden. Das Management-Plugin liefert neben der Möglichkeit, den Payload von Paketen auszulesen, auch Informationen über aktuelle Paket-Raten sowie den Status der einzelnen Channels und Consumer in Echtzeit (Abbildung 4.44 und 4.45).



The screenshot shows the 'Queues' section of the RabbitMQ Management-Plugin. At the top, it says 'Queues' and 'All queues (1)'. Below that is a 'Pagination' section with 'Page 1 of 1' and a 'Filter' input field. The main table has three sections: 'Overview', 'Messages', and 'Message rates'. The 'Overview' section includes columns for Virtual host, Name, Type, Features, State, Ready, Unacked, Total, incoming, deliver / get, and ack. The 'Messages' section shows the current message counts. The 'Message rates' section shows the current message delivery and acknowledgement rates. The 'maptickets' queue is listed in the table with the following details:

Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	maptickets	classic	D Args	running	88	64	152	0.00/s	7.4/s	7.4/s

Abbildung 4.44.: RabbitMQ Management-Plugin Queues



Abbildung 4.45.: RabbitMQ Management-Plugin Queued Messages

4.3.6. Worker

Der Worker stellt die Einheit zur Berechnung der jeweiligen Map bzw. der Chunks dar. Nach dem Start des Dienstes wird eine Verbindung zum RabbitMQ-Dienst aufgebaut und auf den Channel *maptickets* gehorcht. Bei Eingang eines Tickets (siehe Abschnitt 4.3.5) wird eine Anfrage mit der im Ticket enthaltenen Chunk-ID an den Hub-Service gesendet, wodurch der Worker an die jeweiligen Daten des zu berechnenden Chunks gelangt. Sobald der Worker alle nötigen Daten erhalten hat, startet die Berechnung des Chunk-Inhalts mit Hilfe des WFC-Algorithmus (siehe auch Abschnitt 4.2). Sobald die Berechnungen durchgeführt und somit die Generierung des Chunk-Inhalts abgeschlossen ist, werden die Daten an den Hub gesendet, wo der entsprechende Chunk anhand der ID aktualisiert wird. Zusätzlich wird die benötigte Zeit zum Berechnen des Chunks an den Timekeeper-Service gesendet.

Dockerfile

Abbildung 4.46 zeigt die für den Service benutzte Dockerfile. Der Service benutzt die Alpine-Version des offiziellen Python-Images. Alle Dateien im selben Ordner werden in den Container übertragen und letztlich wird das Worker-Skript ausgeführt.

```
1  FROM python:alpine3.19
2  RUN pip install requests pika pandas
3  ADD .
4  CMD ["python", "worker.py"]
```

Abbildung 4.46.: Worker Dockerfile

Callback

In Abbildung 4.47 und 4.48 ist die Callback-Funktion zu sehen, welche ausgeführt wird, sobald eine Nachricht durch den RabbitMQ-Service erhalten wird. Die Funktion erhält 4 Parameter. Der Kanal, auf dem die Nachricht empfangen wurde, die Messaging-Methode, die Eigenschaften der Nachricht, welche allerdings in dieser Funktion nicht benötigt werden, und den *body* bzw. Inhalt der Nachricht. Im ersten Schritt der Funktion wird eine POST-Anfrage an den HUB-Service geschickt. Diese Anfrage bezieht sich auf den jeweiligen Chunk, der mit dieser Nachricht einhergeht. Das Ergebnis wird anschließend in einer Variable gespeichert und enthält unter anderem die Integer-Werte der Potenziale innerhalb dieses Chunks. Mit dem Entschlüsseln des Inhalts dieser Variablen werden vier weitere Variablen extrahiert. Diese sind die Map-ID, die Chunk-ID, die Entropietoleranz und die Liste der Potenziale. Hier wird nun ein

Zeitstempel für den Start der eigentlichen Berechnung gesetzt. Nun werden die von der in dieser Arbeit entwickelten Bibliothek benötigten Variablen an diese übergeben. Als nächstes wird die in Abbildung 4.9 beschriebene Funktion in einer Schleife angewendet, bis diese *True* zurückgibt (siehe auch 4.2). Hiernach ist die Berechnung beendet. Nun wird der berechnete Chunk durch eine POST-Anfrage an den Hub zurückgesendet. Hier wird jetzt der Zeitstempel für das Ende der Berechnung gesetzt. Nachfolgend wird so die Länge der Berechnungszeit ermittelt, welche anschließend durch die Funktion *sendChunkTimes()* (Abbildung 4.51, siehe auch Abschnitt 4.3.6) an den Timekeeper-Service übermittelt wird. Schlussendlich wird die Bearbeitung des Chunks bestätigt, indem ein Bestätigungssignal in den Channel des RabbitMQ-Dienstes gesendet wird.

```

35     def callback(ch, method, properties, body):
36         print("[message received]")
37         finished = False
38         print("[request ticket]")
39         chunk = requests.get(huburl+"/getMapChunkByChunkID/"+body.decode())
40         print("[set map]")
41         chunk = json.loads(chunk.content.decode())
42         mapID = chunk[0]
43         chunkID = chunk[1]
44         mapdata = json.loads(chunk[5])
45
46         startTime = datetime.now()
47         wave.map = mapdata
48         print("[set entropy tolerance]")
49         wave.entropyTolerance = chunk[4]
50         print("[set number of tiles]")
51         wave.numberoftiles = (len(mapdata[0]),len(mapdata))
52         print("[start algorithm]")
53         while not finished:
54             if (not finished):
55                 finished = wave.algorithmStep()
56         print("[finished]")

```

Abbildung 4.47.: Worker callback() 1

Imports, URLs, Rabbit Connection

In Abbildung 4.49 sind die benötigten Imports für den Worker-Service und die Adressen der Services zu sehen, mit denen kommuniziert werden muss. Zusätzlich wird eine Verbindung zu dem RabbitMQ-Service aufgebaut.

```
58     result = requests.post(huburl+"/updateChunkByID",
59                             json = json.dumps({"chunkID":body.decode(),
60                                     "content":wave.map}))
61
62     endTime = datetime.now()
63     # calculate chunkDuration
64     chunkDuration = int((endTime - startTime).total_seconds()*1000)
65     # send chunk times to timekeeper
66     sendChunkTimes(mapID, chunkID, startTime, endTime, chunkDuration)
67     ch.basic_ack(delivery_tag=method.delivery_tag)
```

Abbildung 4.48.: Worker callback() 2

```
1 import wave
2 import json
3 import pika
4 import requests
5 import uuid
6 from datetime import datetime
7
8
9 # WORKER SERVICE
10
11 # URLs
12 huburl = "http://wfchub:5002"
13 timekeeperurl = "http://wfctimekeeper:6002"
14 rabbithost = "wfcrabbit"
15
16 # RABBITMQ CONNECTION
17 connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbithost))
18 channel = connection.channel()
19 channel.queue_declare(queue='maptickets', durable=True)
```

Abbildung 4.49.: Worker Imports, URLs, Rabbit Connection

Main

In Abbildung 4.50 zu sehen ist die Schleife, welche kontinuierlich vom Service ausgeführt wird. Die Bibliothek des RabbitMQ-Services vereinfacht diesen Prozess enorm. Hier wird nur deklariert, wie der Service sich bei Start verhalten soll und von welcher Queue die Nachrichten konsumiert werden sollen. Zusätzlich wird festgelegt, welche Funktion auf diese Nachrichten angewendet werden soll, in diesem Fall ist dies der Callback (siehe 4.3.6). Anschließend kann durch die Bibliothek eine Endlos-Schleife begonnen werden, welche kontinuierlich Nachrichten konsumiert und währenddessen den Main-Thread blockiert.

```
70     print("WORKER SERVICE")
71     print("-----")
72     channel.basic_qos(prefetch_count=1)
73     channel.basic_consume(queue="maptickets", on_message_callback=callback)
74     channel.start_consuming()
```

Abbildung 4.50.: Worker main()

Send Chunk Times

Abbildung 4.51 zeigt die Funktion, welche die Zeitdaten an den Timerkeeper-Service übermittelt. Dies geschieht durch eine POST-Anfrage an den Service, welche die Map-ID, Chunk-ID, Startzeit, Endzeit und Gesamtlänge übermittelt.

```
24     def sendChunkTimes(mapID, chunkID, startTime, endTime, chunkDuration):
25         print("Sending Times to Time Keeper...")
26         result = requests.post(timekeeperurl+"/saveChunkTime",
27                                json = json.dumps({"mapID":mapID, "chunkID":chunkID,
28                                      "startTime":startTime.isoformat(),
29                                      "endTime":endTime.isoformat(),
30                                      "chunkDuration":chunkDuration}))
31         print("Done")
32         print("")
```

Abbildung 4.51.: Worker sendChunkTimes()

4.3.7. Timekeeper

Der Timekeeper-Service bildet die Anwendung zum Sammeln und Speichern der Berechnungszeiten während eines Generievorgangs. Neben der Gesamtzeit, die benötigt wurde, um eine Map zu erzeugen, wird auch jede Dauer zum Generieren eines einzelnen Chunks festgehalten. Der Timekeeper verfügt innerhalb des Clusters über einen alleinigen Zugang zur Time-DB, in der die Zeitdaten gespeichert werden.

Dockerfile

Abbildung 4.52 zeigt die für den Timekeeper-Service verwendete Dockerfile. Auf Basis eines für Python optimierten Alpine-Betriebssystems wird unter anderem die benötigte Erweiterung, um eine MySQL-Verbindung aufzubauen, installiert. Anschließend werden die verwendeten Dateien in das Image kopiert und der Einstiegspunkt konfiguriert.

```

1  FROM python:alpine3.19
2  RUN pip install mysql-connector-python requests flask
3  ADD .
4  EXPOSE 6002
5  CMD ["python", "-m", "flask", "--app", "timekeeper", "run",
6    "--host=0.0.0.0", "--port", "6002"]

```

Abbildung 4.52.: Timekeeper Dockerfile

Imports und globale Variablen

In Abbildung 4.53 sind die für den Timekeeper-Service notwendigen Python-Imports und globalen Variablen zu sehen. Diese Variablen beinhalten sowohl die Adressen der anderen Services, mit denen kommuniziert werden muss, als auch die verschiedenen verwendeten SQL-Statements.

Routen

Nachdem der Dienst gestartet wurde, bietet er folgende Routen an:

- / (**homepage**): Zeigt die Homepage des Dienstes (keine Funktion)
- /**isComplete**/**<uuid>**: Nimmt eine Map-ID entgegenn und antwortet mit *True* oder *False*, abhängig davon, ob die übergebene Map vollständig berechnet wurde
- /**saveMapTime**: Speichert die Messung einer Gesamtzeit in der Datenbank

```

1  import mysql.connector
2  import uuid
3  import json
4  import requests
5  from flask import Flask, request
6  from datetime import datetime
7
8
9  # GLOBAL VARIABLES
10 timedbhost = "timedb"
11 dbhost = "wfcdb"
12 sqlMapInsert = "INSERT INTO mapTimes (" \
13     "mapID, mapSize, chunkCount, numberOfWorkers, startTime, endTime, " \
14     "totalDuration) VALUES (%s, %s, %s, %s, %s, %s, %s);"
15 sqlChunkInsert = "INSERT INTO chunkTimes (" \
16     "mapID, chunkID, startTime, endTime, chunkDuration) " \
17     "VALUES (%s, %s, %s, %s, %s);"
18 sqlGetMapStartTime = "SELECT startTime FROM mapTimes WHERE mapID = %s"
19 sqlUpdate = "UPDATE mapTimes SET endTime = %s, totalDuration = %s WHERE mapID = %s"
20 sqlCheckMapByID = "SELECT computed FROM mapchunks WHERE mapID = %s;"
21 sqlGetChunkEndTimes = "SELECT endTime FROM chunkTimes WHERE mapID = %s;"
```

Abbildung 4.53.: Timekeeper Imports und globale Variablen

- **/updateMapTime:** Aktualisiert eine vorhandene Gesamtzeit
- **/saveChunkTime:** Speichert die Messung einer Chunk-Zeit in der Datenbank

Die Abbildung 4.54 zeigt die ersten zwei Routen für den HTTP-POST-Service. Die erste, „/“ ist die Homepage für den Service, wenn dieser im Browser aufgerufen wird. Die zweite Route ruft eine Funktion auf, an welche über die URL-Parameter eine UUID übergeben wird. Für diese ID wird in der Datenbank kontrolliert, ob die dazugehörige Map bereits fertig berechnet wurde. Anschließend wird ein Boolean zurückgegeben, welcher für den Fall, dass die Map fertig berechnet wurde, *True* - andernfalls *False* enthält.

In Abbildung 4.55 zu sehen ist die dritte Route des Timerkeeper POST-Services */saveMapTime*. An diese werden über die POST-Daten sieben verschiedene Werte übergeben. Diese sind die Startzeit der Berechnung, die UUID der Map, die Kantenlänge der Map, die Gesamtanzahl der Abschnitte, in die die Map zerteilt wurde, die Anzahl der für diese Berechnung verwendeten Worker-Instanzen und die Endzeit sowie die Gesamtdauer der Berechnung. Diese Werte werden anschließend durch einen SQL-Befehl in eine Tabelle der TimeDB-Datenbank geschrieben.

Abbildung 4.56 zeigt die Funktion *updateMapTime()*, welche eine Zeit in der TimeDB-Datenbank updatet. Über die POST-Daten werden die UUID der Map und die Endzeit

```

24     app = Flask(__name__)
25
26     # TIMEKEEPER SERVICE PATHS
27
28     @app.route("/")
29     def showHome():
30         return "TIME KEEPER SERVICE"
31
32
33     @app.route("/isMapComplete/<uuid:ID>", methods=["GET"])
34     def isMapComplete(ID):
35         database = mysql.connector.connect(host=dbhost, database="maps",
36                                              user="root", password="root")
37         dbCursor = database.cursor()
38
39         dbCursor.execute(sqlCheckMapByID, (str(ID),))
40         result = dbCursor.fetchall()
41         database.disconnect()
42
43         completed = True
44         for mapchunkCompleted in result:
45             if not mapchunkCompleted:
46                 completed = False
47
48         return completed

```

Abbildung 4.54.: Timekeeper Routes 1

```

51     @app.route("/saveMapTime", methods=["POST"])
52     def saveMapTime():
53         database = mysql.connector.connect(host=timedbhost, database="times",
54                                              user="root", password="root")
55         dbCursor = database.cursor()
56         data = json.loads(request.json)
57         startTime = datetime.fromisoformat(data["startTime"])
58         valuesToInsert = (data["mapID"], data["mapSize"], data["chunkCount"],
59                           data["numberOfWorkers"], startTime, data["endTime"],
60                           data["totalDuration"])
61         dbCursor.execute(sqlMapInsert, valuesToInsert)
62         database.commit()
63         print("saved maptime in db")
64         database.disconnect()
65         return "done"

```

Abbildung 4.55.: Timekeeper Routes 2

der Berechnung an die Funktion übergeben. Anschließend wird aus der Datenbank die Startzeit der aktuellen Berechnung abgefragt. Mit dieser und der übergebenen Endzeit

wird anschließend die Gesamtdauer der Berechnung bestimmt. Diese Daten werden nun mit einem vorbereiteten SQL-Update-Befehl in die Datenbank geschrieben.

```

68     def updateMapTime(mapID, endTime):
69         database = mysql.connector.connect(host=timedbhost, database="times",
70                                             user="root", password="root")
71         dbCursor = database.cursor()
72         data = json.loads(request.json)
73
74         valuesToFetch = (mapID,)
75         dbCursor.execute(sqlGetMapStartTime, valuesToFetch)
76         row = dbCursor.fetchone()
77         #if row is not None:
78         totalDuration = int((endTime-row[0]).total_seconds()*1000)
79         valuesToUpdate = (endTime, totalDuration, mapID)
80         dbCursor.execute(sqlUpdate, valuesToUpdate)
81         database.commit()
82         print("mptime updated")
83         database.disconnect()
84         return "done"

```

Abbildung 4.56.: Timekeeper Routes 3

In Abbildung 4.57 zu sehen ist die vierte Route des Timekeeper POST-Services `/saveChunkTime`, an welche fünf Werte übergeben werden. Die UUID der Map, welche zu dem aktuellen Chunk gehört, die UUID des Chunks, die Startzeit der Berechnung des Chunks, die Endzeit der Berechnung des Chunks und die Gesamtlänge der Berechnung dieses Chunks. Anschließend werden diese Daten mit einem SQL-Insert-Befehl in die entsprechende Tabelle in der TimeDB-Datenbank geschrieben. Außerdem findet hier eine Kontrolle statt, für den Fall, dass dies der letzte Chunk der aktuellen Map ist und die Map somit fertig berechnet wurde. Wenn dies der Fall ist, wird die späteste Endzeit aller Chunks in dieser Map herausgesucht. Diese wird nun an `updateMapTime()` übergeben.

```
86     @app.route("/saveChunkTime", methods=["POST"])
87     def saveChunkTime():
88         database = mysql.connector.connect(host=timedbhost, database="times",
89                                              user="root", password="root")
90         dbCursor = database.cursor()
91         data = json.loads(request.json)
92         valuesToInsert = (data["mapID"], data["chunkID"],
93                            datetime.fromisoformat(data["startTime"]),
94                            datetime.fromisoformat(data["endTime"]),
95                            data["chunkDuration"])
96         dbCursor.execute(sqlChunkInsert, valuesToInsert)
97         database.commit()
98         print("saved chunktime in db")
99         database.disconnect()
100
101    if isMapComplete(data["mapID"]):
102        database = mysql.connector.connect(host=timedbhost, database="times",
103                                              user="root", password="root")
104        dbCursor = database.cursor()
105        dbCursor.execute(sqlGetChunkEndTimes, (data["mapID"],))
106        resultChunkEndTimes = dbCursor.fetchall()
107
108        lastChunkEndTime = datetime.min
109        for endTime in resultChunkEndTimes:
110            if endTime[0] > lastChunkEndTime:
111                lastChunkEndTime = endTime[0]
112
113        database.disconnect()
114        updateMapTime(data["mapID"], lastChunkEndTime)
115
116    return "done"
```

Abbildung 4.57.: Timekeeper Routes 4

4.3.8. Time-DB

In der Time-DB werden alle Zeiten, die zum Generieren von Chunks und Maps benötigt werden, gespeichert. Die Datenbank besteht aus zwei Tabellen. Die erste Tabelle *mapTimes* dient zum Ablegen der Gesamtzeiten, also ab dem Start des Generierungsvorgangs bis zum Erhalt der vollständigen Map, und beinhaltet folgende Attribute:

- **mapID**: Eindeutige ID der Map
- **mapSize**: Kantenlänge der Map in Tiles
- **chunkCount**: Anzahl der Chunks, in die die Map unterteilt wurde
- **numberOfWorkers**: Anzahl der Worker, die an der Berechnung beteiligt waren
- **startTime**: Zeitstempel beim Start des Vorgangs
- **endTime**: Zeitstempel beim Ende des Vorgangs
- **totalDuration**: Dauer für die Berechnung der Map

Die zweite Tabelle *chunkTimes* dient zum Ablegen der Zeiten, die zur Berechnung der einzelnen Chunks benötigt wurden, und beinhaltet folgende Attribute:

- **mapID**: Identifiziert die Map, zu der der berechnete Chunk gehört und stellt einen Fremdschlüssel mit Referenz auf die *mapTimes*-Tabelle dar
- **chunkID**: Eindeutige ID des Chunks
- **startTime**: Zeitstempel beim Start des Vorgangs
- **endTime**: Zeitstempel beim Ende des Vorgangs
- **chunkDuration**: Dauer für die Berechnung des Chunks

Der Timekeeper-Service (siehe 4.3.7) verfügt innerhalb des Clusters über eine alleinige Verbindung zur Time-DB.

Abbildung 4.58 zeigt die für den Time-DB-Service verwendete Dockerfile. Auf Basis des von Oracle zur Verfügung gestellten MySQL-Images werden die Dateien zum Initialisieren der Datenbank in das Verzeichnis kopiert.

```
1  FROM mysql
2  ADD ./init.sql /docker-entrypoint-initdb.d/
```

Abbildung 4.58.: Time-DB Dockerfile

In Abbildung 4.59 zu sehen ist die *init.sql*-Datei, welche bei dem Build-Verfahren des Containers ausgeführt wird. Diese enthält das Erstellen der *times*-Datenbank und deren zwei Tabellen, *mapTimes* und *chunkTimes*.

```
1  CREATE DATABASE IF NOT EXISTS times;
2  USE times;
3  CREATE TABLE IF NOT EXISTS mapTimes (
4      mapID VARCHAR(36),
5      mapSize INT,
6      chunkCount INT,
7      numberOfWorkers INT,
8      startTime DATETIME(3),
9      endTime DATETIME(3),
10     totalDuration INT,
11     PRIMARY KEY (mapID)
12 );
13
14 CREATE TABLE IF NOT EXISTS chunkTimes (
15     mapID VARCHAR(36),
16     chunkID VARCHAR(36),
17     startTime DATETIME(3),
18     endTime DATETIME(3),
19     chunkDuration INT,
20     PRIMARY KEY (mapID, chunkID),
21     FOREIGN KEY (mapID) REFERENCES mapTimes(mapID)
22 );
```

Abbildung 4.59.: Time-DB Init

4.4. Visualisierung

4.4.1. Maploader

Der Maploader ist das Programm, welches zum Anzeigen einer Map verwendet wird (Abbildung 4.60). Nachdem die Anwendung gestartet wurde, gibt der Benutzer die gewünschte Map-ID ein (Abbildung 4.61). Die Anwendung stellt sodann eine Verbindung zum Hub-Service her und fragt dort die entsprechende Map an. Nachdem die Daten übertragen wurden, wird die Map auf einer grafischen Oberfläche ausgegeben (siehe Abschnitt 4.4.2).

```
1  import os
2  import requests
3  import visualization
4
5  HUBADDR = "192.168.178.56"
6  HUBPORT = "31002"
7
8  huburl = "http://"+HUBADDR+":"+HUBPORT+"/"
9
10 mapID = "-1"
11
12 # MAP VISUALIZATION
13
14 def showmenu():
15     os.system('cls')
16     print("")
17     print("-----")
18     print("| MAP VISUALIZATION |")
19     print("-----")
20     print("")
21     mapID = input("MAP#: ")
22     return mapID
23
24
25 mapID = showmenu()
26 fullmap = requests.get(huburl+"/getMapByID/"+mapID).json()
27
28 visualization.showmap(fullmap)
```

Abbildung 4.60.: Maploader Applikation Code

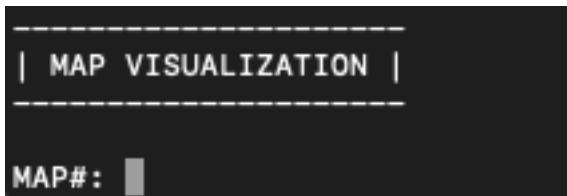


Abbildung 4.61.: Maploader Applikation

4.4.2. Visualization-Bibliothek

Imports

In Abbildung 4.62 sind die für die Visualisierung benötigten Imports und statischen Variablen zu sehen. Darunter fällt auch die *wave*- sowie die *wavefunctionlookup*-Bibliothek, welche in Abschnitt 4.2 behandelt werden. Des Weiteren wird die Funktion *requestRestrictions()* ausgeführt, welche die Restriktionen und Regeln beim Manager-Service anfragt.

```
1 import pygame
2 import wavefunctionlookup as wfl
3 from wave import numberOfOnes
4
5
6 displaySizeX = 512
7 displaySizeY = 512
8
9 wfl.requestRestrictions()
```

Abbildung 4.62.: Visualization Imports

Initialize Tiles

Abbildung 4.63 zeigt die Funktion des Maploader-Skripts, welches die benötigten Sprites aus PNG-Dateien lädt und skaliert.

Select Image

In Abbildung 4.64 ist die Funktion zu sehen, welche einen Canvas mit dem benötigten Bild zurückgibt. Zuerst wird kontrolliert, ob das übergebene Potenzial mehr als eine Eins enthält (siehe auch Abschnitt 4.2). Wenn dies der Fall ist, ist das Potenzial

```

12  def initializeTiles(scaleValue):
13      global tileImg
14      global grassImg
15      global waldImg
16      global kuhImg
17      global strandImg
18      global wasserImg
19      global fischImg
20      global bergImg
21      global bergschneeImg
22      global schneemannImg
23      tileImg = pygame.transform.scale(
24          pygame.image.load("TILES/tile.png"), scaleValue)
25      grassImg = pygame.transform.scale(
26          pygame.image.load("TILES/grass.png"), scaleValue)
27      waldImg = pygame.transform.scale(
28          pygame.image.load("TILES/wald.png"), scaleValue)
29      kuhImg = pygame.transform.scale(
30          pygame.image.load("TILES/kuh.png"), scaleValue)
31      strandImg = pygame.transform.scale(
32          pygame.image.load("TILES/strand.png"), scaleValue)
33      wasserImg = pygame.transform.scale(
34          pygame.image.load("TILES/wasser.png"), scaleValue)
35      fischImg = pygame.transform.scale(
36          pygame.image.load("TILES/fisch.png"), scaleValue)
37      bergImg = pygame.transform.scale(
38          pygame.image.load("TILES/berg.png"), scaleValue)
39      bergschneeImg = pygame.transform.scale(
40          pygame.image.load("TILES/bergschnee.png"), scaleValue)
41      schneemannImg = pygame.transform.scale(
42          pygame.image.load("TILES/schneemann.png"), scaleValue)

```

Abbildung 4.63.: Visualization initializeTiles()

noch nicht komplett kollabiert worden. In diesem Fall sollen alle benötigten Bilder transparent gemacht und aufeinander gestapelt werden. So soll der Nutzer auf einen Blick sehen können, welche Zustände ein angezeigtes Potenzial noch annehmen kann. Um dies zu erreichen, wird zuerst ein Canvas angelegt und mit der Farbe Weiß gefüllt. Im nächsten Schritt wird jede Stelle in der Binärrepräsentation des Potenzials geprüft, und wenn diese eine Eins ist, wird das entsprechende Bild transparent gemacht und auf den existierenden Canvas gestapelt. Um das Bild für die entsprechende Stelle zu laden, wird diese Funktion rekursiv ausgeführt. Dies führt nicht zu Problemen, weil das hier übergebene Potenzial nur eine Eins hat und nur einem Bild entspricht. Wenn dieser Prozess fertig ist, werden die gestapelten Bilder mit dem Canvas zurückgegeben. Für den Fall, wie oben beschrieben, dass das Potenzial nur eine Eins enthält, wird

das übergebene Potenzial durch ein binäres *UND* mit allen Werten für die jeweiligen Bilder in der binären Lookup-Tabelle verglichen. Ist dieses gleich 0, entspricht dieses Potenzial diesem Bild und kann somit zurückgegeben werden.

```

45  def selectImage(tile, scaleValue):
46      if (numberOfOnes(tile) > 1):
47          canvas = pygame.Surface(scaleValue,pygame.SRCALPHA)
48          canvas.fill((255,255,255,255))
49
50          for e in range (0,8):
51              if (tile&(2**e) != 0):
52                  tempImg = selectImage(2**e, scaleValue).copy()
53                  tempImg.set_alpha(128)
54                  canvas.blit(tempImg,(0,0),special_flags=pygame.BLEND_RGBA_MULT)
55          return canvas
56
57      if (tile&wfl.binaryLookUpTable["grass"] != 0): return grassImg
58      if (tile&wfl.binaryLookUpTable["wald"] != 0): return waldImg
59      if (tile&wfl.binaryLookUpTable["kuh"] != 0): return kuhImg
60      if (tile&wfl.binaryLookUpTable["strand"] != 0): return strandImg
61      if (tile&wfl.binaryLookUpTable["wasser"] != 0): return wasserImg
62      if (tile&wfl.binaryLookUpTable["fisch"] != 0): return fischImg
63      if (tile&wfl.binaryLookUpTable["berg"] != 0): return bergImg
64      if (tile&wfl.binaryLookUpTable["bergschnee"] != 0): return bergschneeImg
65      if (tile&wfl.binaryLookUpTable["schneemann"] != 0): return schneemannImg

```

Abbildung 4.64.: Visualization selectImage()

Abbildung 4.65 zeigt die Funktion, welche zur eigentlichen Visualisierung der Tile-Map führt. Zuerst werden die für die Pygame-Engine benötigten Funktionen ausgeführt und die Optionen konfiguriert. Im nächsten Schritt werden die einzelnen Potenziale durch die Funktion *initializeTiles()* (siehe 4.4.2) als Tiles initialisiert. Nun geht die Funktion in eine Endlos-Schleife. Diese ist durch die Pygame-Engine auf 60 FPS limitiert und kann mit dem Druck der ESC-Taste beendet werden. Darauffolgend wird das Fenster mit der Farbe Weiß als Hintergrundfarbe gefüllt. Im nächsten Schritt wird die komplette zweidimensionale Tile-Map durchlaufen und jedes Potenzial wird einzeln der Funktion *selectImage()* (siehe 4.4.2), zusammen mit der gewünschten Skalierung, übergeben. Das Bild wird an der Stelle in das Fenster gezeichnet, welches mit der Position in der Tile-Map korrespondiert. Letztlich wird mit der *flip*-Funktion der Pygame-Engine der angelegte Canvas wirklich in das Fenster gezeichnet und der Vorgang kann erneut beginnen.

```
68  def showmap(mapdata):
69      pygame.init()
70      screen = pygame.display.set_mode((displaySizeX, displaySizeY))
71      clock = pygame.time.Clock()
72      running = True
73      scaleValue = (displaySizeX/len(mapdata[0]),displaySizeY/len(mapdata))
74      initializeTiles(scaleValue)
75      while running:
76          clock.tick(60) # limits FPS to 60
77
78          # polls for events
79          for event in pygame.event.get():
80              if event.type == pygame.QUIT:
81                  running = False
82          # fills the screen with a color
83          screen.fill("white")
84
85          # RENDERING
86          for y in range (0,len(mapdata[0])):
87              for x in range (0,len(mapdata)):
88                  screen.blit(selectImage(mapdata[y][x], scaleValue),
89                              (scaleValue[0]*x,scaleValue[1]*y))
90      pygame.display.flip()
```

Abbildung 4.65.: Visualization showmap()

4.5. Kubernetes

In diesem Abschnitt wird die Installation und Konfiguration des Kubernetes-Clusters anhand der innerhalb des Projekts gefertigten Skripte behandelt. Um das Cluster in Betrieb zu nehmen, sind verschiedene Schritte notwendig, welche zusätzlich auch ausführlich in der Readme-Datei des Git-Repositorys (siehe Anhang A.1) beschrieben werden. Ziel ist es, Docker, Containerd und Kubernetes zu installieren und zu konfigurieren sowie ein Calico-Netzwerk für das Cluster bereitzustellen.

Das Skript liegt in zwei Versionen vor, je eine für die Manager-Node und eine für die Worker-Nodes. Die Konfiguration der Manager-Node beinhaltet zusätzlich die Generierung eines sogenannten *Join-Tokens*, welcher auf die Worker-Nodes transferiert werden muss, um sie dem durch die Manager-Node erstellten Cluster hinzuzufügen.

4.5.1. Installation und Konfiguration

Die Abbildungen 4.66 und 4.67 zeigen den Teil des Kubernetes-Installationsskripts, welcher universell auf jeder Maschine ausgeführt wird. Zunächst wird Docker als Grundlage zur Containerverwaltung installiert und der Swap-Speicher deaktiviert, da Kubernetes nicht mit aktivem Swap funktioniert. Zusätzlich wird der Swap-Eintrag in `/etc/fstab/` auskommentiert, damit Swap auch nach einem Neustart des Systems weiterhin deaktiviert ist. Anschließend werden die Kernel-Module *overlay* (für Overlay-Dateisysteme) und *br netfilter* (für den Netzwerkverkehr von Pods) aktiviert und gestartet. Dazu wird IPv4-Forwarding aktiviert, damit Pods Pakete weiterleiten können.

Im nächsten Schritt werden Tools für HTTPS-Zugriffe und Zertifikate installiert, damit das Kubernetes-Repository aufgerufen werden kann. Anschließend werden die Kernkomponenten von Kubernetes *kubelet*, *kubeadm* und *kubectl* (siehe auch Kapitel 2.4) installiert, bevor im letzten Schritt *containerd* konfiguriert - und zusammen mit dem *kubelet*-Service neugestartet wird (siehe Abbildung 4.67).

Der nun folgende Abschnitt (zu sehen in Abbildung 4.68) wird ausschließlich auf der Manager-Node ausgeführt. Zuerst werden von *kubeadm* alle benötigten Images zum Initialisieren einer Manager-Node heruntergeladen. Danach kann das Cluster, unter Angabe des Subnetzes für Pods, initialisiert werden. Die Ausgabe des Befehls beinhaltet den *Join-Token* und wird in einer separaten Datei gespeichert. Im nächsten Schritt wird *kubectl* für den aktuellen Benutzer konfiguriert, um die Verwaltung des Clusters zu ermöglichen. Nun wird das Calico Netzwerk-Plugin heruntergeladen, installiert und an das vorher durch *kubeadm init* erstellte Subnetz angepasst. Dazu werden Informationen aus der *custom-resources* YAML-Datei (Abbildung 4.69) abgerufen. Im

```

1  #!/bin/bash
2  sudo apt install docker.io -y
3  sudo swapoff -a
4  sudo sed -i '/swap/s/^/#/' /etc/fstab
5  sudo swapon --show
6  cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
7  overlay
8  br_netfilter
9  EOF
10
11 sudo modprobe overlay
12 sudo modprobe br_netfilter
13 cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
14 net.bridge.bridge-nf-call-iptables = 1
15 net.bridge.bridge-nf-call-ip6tables = 1
16 net.ipv4.ipforward = 1
17 EOF

```

Abbildung 4.66.: Kubernetes Install Script 1

```

19 sudo sysctl --system
20 sudo apt install curl ca-certificates apt-transport-https -y
21 curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key
22 | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
23 sudo apt-get install -y apt-transport-https ca-certificates curl gnupg
24 sudo chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg
25 echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
26 https://pkgs.k8s.io/core:/stable:/v1.30/deb/ \
27 | sudo tee /etc/apt/sources.list.d/kubernetes.list
28 sudo chmod 644 /etc/apt/sources.list.d/kubernetes.list
29 sudo apt update
30 sudo apt install kubelet kubeadm kubectl -y
31
32 sudo mkdir /etc/containerd
33 sudo sh -c "containerd config default > /etc/containerd/config.toml"
34 sudo sed -i 's/ SystemdCgroup = false/ SystemdCgroup = true/' \
35 | /etc/containerd/config.toml
36 sudo systemctl restart containerd.service
37 sudo systemctl restart kubelet.service
38 sudo systemctl enable kubelet.service

```

Abbildung 4.67.: Kubernetes Install Script 2

letzten Schritt wird der in der Datei *token.sh* gespeicherte Join-Token extrahiert und in eine neue Datei *jontoken.sh* gespeichert. Die neue Datei ist so formatiert, dass sie auf jede beliebige Worker-Node transferiert - und dort direkt ausgeführt werden kann, um sie dem Cluster hinzuzufügen.

```

39 #MASTER PART
40 sudo kubeadm config images pull
41 sudo kubeadm init --pod-network-cidr=10.10.0.0/16 > token.sh
42 mkdir -p $HOME/.kube
43 sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
44 sudo chown $(id -u):$(id -g) $HOME/.kube/config
45
46 kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/
47 | | | | manifests/tigera-operator.yaml --validate=false
48 curl https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/
49 | | | custom-resources.yaml -O
50 ls -l
51 sed -i 's/cidr: 192\.168\.0\.0/16/cidr: 10.10.0.0/16/g' custom-resources.yaml
52 kubectl create -f custom-resources.yaml --validate=false
53 sudo tail -n 2 token.sh > jointoken.sh

```

Abbildung 4.68.: Kubernetes Install Script Manager

```

1 # This section includes base Calico installation configuration.
2 apiVersion: operator.tigera.io/v1
3 kind: Installation
4 metadata:
5   name: default
6 spec:
7   # Configures Calico networking.
8   calicoNetwork:
9     # Note: The ipPools section cannot be modified post-install.
10    ipPools:
11      - blockSize: 26
12        cidr: 10.10.0.0/16
13        encapsulation: VXLANCrossSubnet
14        natOutgoing: Enabled
15        nodeSelector: all()
16
17 ---
18
19 # This section configures the Calico API server.
20 apiVersion: operator.tigera.io/v1
21 kind: APIServer
22 metadata:
23   name: default
24 spec: {}

```

Abbildung 4.69.: Custom Resources YAML

4.5.2. Deployment

Das Kubernetes-Deployment wird durch eine YAML-Datei beschrieben. Durch das Ausführen dieser Datei mit Hilfe von *kubectl* werden alle benötigten Dienste der Software ausgerollt. Der Start beinhaltet standardmäßig 2 Worker-Instanzen (*replicas*) welche anschließend beliebig skaliert werden können. Der genaue Vorgang zum Starten des Deployments sowie zum Skalieren der Worker wird in der Readme-Datei des Git-Repositorys (Anhang A.1) Schritt für Schritt beschrieben.

Für jeden Service wird zusätzlich das entsprechende Deployment konfiguriert und beinhaltet neben dem im Cluster verwendeten Namen auch das verwendete Image sowie den zu verwendenden Port, gegebenenfalls *Nodeport* und die Anzahl der Replicas. Bei Services, welche eine Datenbank enthalten, wird außerdem ein *Persistent Volume* zur dauerhaften Speicherung der Daten angelegt und die Umgebungsvariablen der Datenbank (Nutzername, Passwort, verwendete Datenbank) werden konfiguriert (siehe Abbildung 4.70 bis 4.87).

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: timedb
5    labels:
6      app: timedb
7  spec:
8    type: NodePort
9    ports:
10   - port: 3306
11     name: dtimedbport
12     protocol: TCP
13     targetPort: 3306
14     nodePort: 31006
15   selector:
16     app: timedb
17 ---
```

Abbildung 4.70.: Deployment wfcdeploy.yaml 1

```
18 apiVersion: v1
19 kind: PersistentVolumeClaim
20 metadata:
21   name: timedb-pvc
22 spec:
23   accessModes:
24     - ReadWriteOnce
25   resources:
26     requests:
27       storage: 1Gi
28 ---
29 apiVersion: v1
30 kind: PersistentVolume
31 metadata:
32   name: timedb-storage
33 spec:
34   capacity:
35     storage: 1Gi
36   accessModes:
37     - ReadWriteOnce
38   persistentVolumeReclaimPolicy: Retain
39   hostPath:
40     path: "/mnt/data" # volume location
41 ---
```

Abbildung 4.71.: Deployment wfcdeploy.yaml 2

```
42 apiVersion: apps/v1
43 kind: StatefulSet
44 metadata:
45   name: timedb-deployment
46 spec:
47   replicas: 1
48   selector:
49     matchLabels:
50       app: timedb
51   template:
52     metadata:
53       labels:
54         app: timedb
```

Abbildung 4.72.: Deployment wfcdeploy.yaml 3a

```
55   spec:
56     volumes:
57       - name: timedb-storage
58         persistentVolumeClaim:
59           claimName: timedb-pvc
60     containers:
61       - name: timedb-container
62         image: ppropfen/imgtimedb
63         imagePullPolicy: Always
64         env:
65           - name: MYSQL_ROOT_USER
66             value: "root"
67           - name: MYSQL_ROOT_PASSWORD
68             value: "root"
69           - name: MYSQL_USER
70             value: "wfc"
71           - name: MYSQL_PASSWORD
72             value: "wfc"
73           - name: MYSQL_DATABASE
74             value: "times"
75         ports:
76           - containerPort: 3306
77         volumeMounts:
78           - mountPath: /var/lib/mysql
79             name: timedb-storage
80   ---
```

Abbildung 4.73.: Deployment wfcdeploy.yaml 3b

```
81   apiVersion: v1
82   kind: Service
83   metadata:
84     name: wfcdb
85     labels:
86       app: wfcdb
87   spec:
88     ports:
89       - port: 3306
90     selector:
91       app: wfcdb
92   ---
```

Abbildung 4.74.: Deployment wfcdeploy.yaml 4

```
93  apiVersion: apps/v1
94  kind: Deployment
95  metadata:
96    name: wfcdb-deployment
97  spec:
98    replicas: 1
99    selector:
100      matchLabels:
101        app: wfcdb
102    template:
103      metadata:
104        labels:
105          app: wfcdb
106      spec:
107        containers:
108          - name: wfcdb-container
109            image: ppropfen/imgdb
110            imagePullPolicy: Always
111            env:
112              - name: MYSQL_ROOT_USER
113                value: "root"
114              - name: MYSQL_ROOT_PASSWORD
115                value: "root"
116              - name: MYSQL_USER
117                value: "wfc"
118              - name: MYSQL_PASSWORD
119                value: "wfc"
120              - name: MYSQL_DATABASE
121                value: "maps"
122            ports:
123              - containerPort: 3306
124  ---
```

Abbildung 4.75.: Deployment wfcdeploy.yaml 5

```
125  apiVersion: v1
126  kind: Service
127  metadata:
128    name: wfcrabbit
129    labels:
130      app: wfcrabbit
131  spec:
132    ports:
133      - port: 5672
134    selector:
135      app: wfcrabbit
136  ---
```

Abbildung 4.76.: Deployment wfcdeploy.yaml 6

```
137  apiVersion: apps/v1
138  kind: Deployment
139  metadata:
140    name: wfcrabbit-deployment
141  spec:
142    replicas: 1
143    selector:
144      matchLabels:
145        app: wfcrabbit
146    template:
147      metadata:
148        labels:
149          app: wfcrabbit
150    spec:
151      containers:
152        - name: wfcrabbit-container
153          image: rabbitmq:latest
154          imagePullPolicy: Always
155        ports:
156          - containerPort: 5672
157  ---
```

Abbildung 4.77.: Deployment wfcdeploy.yaml 7

```
158 apiVersion: v1
159 kind: Service
160 metadata:
161   name: wfcmanager
162   labels:
163     app: wfcmanager
164 spec:
165   type: NodePort
166   ports:
167     - port: 5000
168       name: wfcmanagerport
169       targetPort: 5000
170       nodePort: 31000
171   selector:
172     app: wfcmanager
173 ---
```

Abbildung 4.78.: Deployment wfcdeploy.yaml 8

```
174 apiVersion: apps/v1
175 kind: Deployment
176 metadata:
177   name: wfcmanager-deployment
178 spec:
179   replicas: 1
180   selector:
181     matchLabels:
182       app: wfcmanager
183   template:
184     metadata:
185       labels:
186         app: wfcmanager
187     spec:
188       containers:
189         - name: wfcmanager-container
190           image: pfrompen/imgmanager
191           imagePullPolicy: Always
192           ports:
193             - containerPort: 5000
194 ---
```

Abbildung 4.79.: Deployment wfcdeploy.yaml 9

```
195  apiVersion: v1
196  kind: Service
197  metadata:
198    name: wfcdistribution
199    labels:
200      app: wfcdistribution
201  spec:
202    type: NodePort
203    ports:
204      - port: 5001
205        name: distributorport
206        protocol: TCP
207        targetPort: 5001
208        nodePort: 31001
209    selector:
210      app: wfcdistribution
211 ---
```

Abbildung 4.80.: Deployment wfcdeploy.yaml 10

```
212  apiVersion: apps/v1
213  kind: Deployment
214  metadata:
215    name: wfcdistribution-deployment
216  spec:
217    replicas: 1
218    selector:
219      matchLabels:
220        app: wfcdistribution
221    template:
222      metadata:
223        labels:
224          app: wfcdistribution
225      spec:
226        containers:
227          - name: wfcdistribution-container
228            image: ppropfen/imgdistribution
229            imagePullPolicy: Always
230            ports:
231              - containerPort: 5001
```

Abbildung 4.81.: Deployment wfcdeploy.yaml 11

```
233   apiVersion: v1
234   kind: Service
235   metadata:
236     name: wfchub
237     labels:
238       app: wfchub
239   spec:
240     type: NodePort
241     ports:
242       - port: 5002
243         name: wfchubport
244         targetPort: 5002
245         nodePort: 31002
246     selector:
247       app: wfchub
248 ---
```

Abbildung 4.82.: Deployment wfcdeploy.yaml 12

```
249   apiVersion: apps/v1
250   kind: Deployment
251   metadata:
252     name: wfchub-deployment
253   spec:
254     replicas: 1
255     selector:
256       matchLabels:
257         app: wfchub
258     template:
259       metadata:
260         labels:
261           app: wfchub
262     spec:
263       containers:
264         - name: wfchub-container
265           image: pfropfen/imghub
266           imagePullPolicy: Always
267           ports:
268             - containerPort: 5002
269 ---
```

Abbildung 4.83.: Deployment wfcdeploy.yaml 13

```
270  apiVersion: v1
271  kind: Service
272  metadata:
273    name: wfcworker
274    labels:
275      app: wfcworker
276  spec:
277    ports:
278      - port: 5003
279    selector:
280      app: wfcworker
281  ---
```

Abbildung 4.84.: Deployment wfcdeploy.yaml 14

```
282  apiVersion: apps/v1
283  kind: Deployment
284  metadata:
285    name: wfcworker-deployment
286  spec:
287    replicas: 2
288    selector:
289      matchLabels:
290        app: wfcworker
291    template:
292      metadata:
293        labels:
294          app: wfcworker
295      spec:
296        containers:
297          - name: wfcworker-container
298            image: ppropfen/imgworker
299            imagePullPolicy: Always
300          ports:
301            - containerPort: 5003
302  ---
```

Abbildung 4.85.: Deployment wfcdeploy.yaml 15

```
303   apiVersion: v1
304   kind: Service
305   metadata:
306     name: wfctimekeeper
307   labels:
308     app: wfctimekeeper
309   spec:
310     type: NodePort
311     ports:
312       - port: 6002
313         name: wfctimekeeperport
314         targetPort: 6002
315         nodePort: 32002
316     selector:
317       app: wfctimekeeper
318 ---
```

Abbildung 4.86.: Deployment wfcdeploy.yaml 16

```
319   apiVersion: apps/v1
320   kind: Deployment
321   metadata:
322     name: wfctimekeeper-deployment
323   spec:
324     replicas: 1
325     selector:
326       matchLabels:
327         app: wfctimekeeper
328     template:
329       metadata:
330         labels:
331           app: wfctimekeeper
332     spec:
333       containers:
334         - name: wfctimekeeper-container
335           image: pfdpropfen/imgtimekeeper
336           imagePullPolicy: Always
337           ports:
338             - containerPort: 6002
```

Abbildung 4.87.: Deployment wfcdeploy.yaml 17

4.6. Parallelisierung

Die Parallelisierung findet im Distributor-Service statt, welcher über eine eigene Implementierung des WFC-Algorithmus in Form der *wave.py*-Bibliothek verfügt. Die genaue Implementierung bzw. die Behandlung der Parallelisierung befinden sich somit in Abschnitt 4.2 (WFC) und 4.3.2 (Distributor).

4.7. Tools

Während des Projektes wurde eine Reihe von Tools erstellt, welche die Entwicklung und den Betrieb der Software vereinfachen. In diesem Abschnitt wird auf die Funktionsweise der einzelnen Tools eingegangen.

4.7.1. Timeextractor

Der Timeextractor ist ein Programm zum Auslesen der Time-DB. Alle in der Datenbank gespeicherten Tabellen werden bei Ausführung des Programms in separate CSV-Dateien exportiert. Dies dient der späteren Evaluation der Zeitdaten.

Imports

Abbildung 4.88 zeigt die für das Timeextractor-Skript benötigten Python-Imports. Darunter die Möglichkeit, eine Verbindung zu einer MySQL-Datenbank aufzubauen, CSV-Dateien zu erstellen und auf Verzeichnisstrukturen zuzugreifen.

```
1 import mysql.connector  
2 import csv  
3 import os
```

Abbildung 4.88.: Timeextractor Imports

Export-Funktion

In Abbildung 4.89 und 4.90 ist die Funktion zu sehen, welche für den Export der Daten der Time-DB dient. Nachdem das Objekt zur Kommunikation mit der Datenbank erstellt ist, werden zuerst alle Namen der in der Datenbank enthaltenen Tabellen angefragt. Im nächsten Schritt wird über alle zurückgegebenen Tabellen-Namen iteriert.

Für jede dieser Tabellen wird eine Anfrage an die Datenbank geschickt, alle in dieser Tabelle enthaltenen Daten zu übermitteln. Diese Daten werden nun zusammen mit dem Namen aller Spalten in eine CSV-Datei geschrieben, welche nach der Tabelle benannt ist.

```
5 def export_database_to_csv(host, user, password, database, port=3306,
6 | | | | | output_dir="output"):
7 conn = mysql.connector.connect(
8 | | host=host,
9 | | user=user,
10 | | password=password,
11 | | database=database,
12 | | port=port
13 )
14 cursor = conn.cursor()
15
16 if not os.path.exists(output_dir):
17 | os.makedirs(output_dir)
18
19 cursor.execute("SHOW TABLES")
20 tables = cursor.fetchall()
```

Abbildung 4.89.: Timeextractor Export-Funktion 1

Main

In Abbildung 4.91 ist die für den Service benötigte Verbindung zur Time-DB und deren Konfiguration zu sehen.

```
22     for (table_name,) in tables:
23         print(f"Exporting {table_name}...")
24
25         cursor.execute(f"SELECT * FROM {table_name}")
26         rows = cursor.fetchall()
27
28         column_names = [desc[0] for desc in cursor.description]
29
30         csv_file_path = os.path.join(output_dir, f"{table_name}.csv")
31
32         # write data to CSV
33         with open(csv_file_path, mode="w", newline="", encoding="utf-8") \
34             as csv_file:
35             writer = csv.writer(csv_file)
36             writer.writerow(column_names) # headers
37             writer.writerows(rows) # data
38
39         print(f"{table_name} exported successfully to {csv_file_path}")
40
41         # close connection
42         cursor.close()
43         conn.close()
44         print("Database export completed.")
```

Abbildung 4.90.: Timeextractor Export-Funktion 2

```
46     # connection to DB
47     export_database_to_csv(host="192.168.178.56",
48                           user="root",
49                           password="root",
50                           database="times",
51                           port=31006)
```

Abbildung 4.91.: Timeextractor Main

4.7.2. messen.py

Dieses Skript wird verwendet, um eine Reihe von automatisierten Messvorgängen durchzuführen und kann direkt über die Konsole zusammen mit einem Argument X gestartet werden, wobei X die Anzahl der Worker-Replicas angibt, für welche die Messreihe durchgeführt werden soll. Für den Vorgang wird jeweils eine Verbindung zu der Hub-DB, der Time-DB und dem RabbitMQ-Service benötigt (siehe Abbildung 4.92). Das Skript greift nun auf die Datei *messreihen.csv* zu (Abbildung 5.1) und arbeitet diese Zeile für Zeile ab. Jede Zeile der Datei enthält die Informationen über die Größe der zu erstellenden Map (Kantenlänge), die Anzahl der Abschnitte, in die sie zerteilt werden soll, und die Anzahl der beteiligten Worker. Für jede Zeile, in der die angegebene Anzahl an Workern mit der als Argument X übergebenen Anzahl übereinstimmt, wird ein Generierungsprozess gestartet. Aus Performance-Gründen werden die Vorgänge mit einer voreingestellten Entropietoleranz von 0 durchgeführt (siehe auch Kapitel 3.2.5 und 4.8.7). Um zu überprüfen, ob eine Generierung abgeschlossen wurde, wird wiederholt eine Anfrage an die Hub-DB gesendet, in der per SQL-Befehl die *computed*-Werte der einzelnen Chunks geprüft werden. Falls alle zu der zu generierenden Map-ID gehörigen Chunks *computed* sind, ist die Map vollständig berechnet (Abbildung 4.94 bis 4.97). Um Verfälschungen der Messungen zu vermeiden, wird nach der Bestätigung der *computed*-Werte zusätzlich geprüft, ob sich noch Nachrichten in der Message-Queue befinden (Abbildung 4.98). Ist dies nicht der Fall, wird mit der nächsten Messung begonnen. Der Abbruch des Vorgangs kann jederzeit durch das Drücken der Taste *X* angefordert (aktuelle Messung wird noch abgeschlossen) - oder durch *Q* erzwungen werden (Vorgang wird direkt beendet) - (Abbildung 4.99).

```
1 import requests
2 from bs4 import BeautifulSoup
3 import mysql.connector
4 import csv
5 import time
6 import threading
7 import keyboard # type: ignore
8 import sys
9
10 baseIP = "139.6.65.27"
11
12 db1Config = {
13     "host": baseIP,
14     "port": 31006,
15     "user": "wfc",
16     "password": "wfc",
17     "database": "times"
18 }
19
20 db2Config = {
21     "host": baseIP,
22     "port": 31007,
23     "user": "wfc",
24     "password": "wfc",
25     "database": "maps"
26 }
27
28 rabbitmqUser = "guest"
29 rabbitmqPassword = "guest"
30 rabbitmqQueue = "maptickets"
31 rabbitmqApiUrl = f"http://{baseIP}:31672/api/queues/%2F/{rabbitmqQueue}"
```

Abbildung 4.92.: messen.py Imports

```
50 numberOfWorkers = int(sys.argv[1])
51
52 csvPath = "messreihen.csv"
53 uuidColumnName = "uuid"
54 dbValueColumnName = "total"
55 statusPollInterval = 5
56 maxWaitTime = 1200
57
58 exitRequested = False
59 immediateExitRequested = False
```

Abbildung 4.93.: messen.py Variablen

```
75     threading.Thread(target=watchForExit, daemon=True).start()
76     threading.Thread(target=watchForImmediateExit, daemon=True).start()
77
78     updatedRows = []
79     with open(csvPath, mode="r", newline="") as file:
80         reader = csv.reader(file)
81         headers = next(reader)
82
83         if uuidColumnName not in headers:
84             headers.append(uuidColumnName)
85         if dbValueColumnName not in headers:
86             headers.append(dbValueColumnName)
87
88         for rowIndex, row in enumerate(reader, start=1):
89             if exitRequested:
90                 updatedRows.append(row)
91                 break
92
93             print("Row: ", row)
94             if int(row[2]) == numberOfRows and (len(row)<7 or row[6].strip() == ""):
95                 payload = {
96                     "var1": str(row[0]),
97                     "var2": str(row[1]),
98                     "var3": "0",
99                     "var4": str(row[2])
100                }
101
102                 response = requests.post(f"http://[{baseIP}]:31000/setRules",
103                                         data=payload)
104                 time.sleep(30)
```

Abbildung 4.94.: messen.py Main 1

```
106
107     try:
108         response = requests.post(f"http://{baseIP}:31001/mapGenerator")
109         if response.ok:
110             soup = BeautifulSoup(response.text, "html.parser")
111             uuidTag = soup.h1
112             uuid = uuidTag.text.strip() if uuidTag else ""
113         else:
114             uuid = ""
115     except Exception as e:
116         print(f"Error during POST for row {rowIndex}: {e}")
117         uuid = ""
118
119     dbValue = ""
120     if uuid:
121         startTime = time.time()
122         while True:
123             if immediateExitRequested:
124                 break
125             try:
126                 conn2 = mysql.connector.connect(**db2Config)
127                 cursor2 = conn2.cursor()
128                 cursor2.execute("SELECT mapID FROM mapchunks " \
129                               "WHERE mapID = %s " \
130                               "GROUP BY mapID " \
131                               "HAVING SUM" \
132                               "(CASE WHEN computed IS NOT TRUE THEN 1 ELSE 0 END) = 0",
133                               (uuid,))
134
135                 statusResult = cursor2.fetchone()
136                 cursor2.close()
137                 conn2.close()
```

Abbildung 4.95.: messen.py Main 2

```
138         if statusResult:
139             print(f"Computation complete for UUID {uuid}")
140
141             print("Waiting for RabbitMQ queue to empty...")
142             while not isQueueEmpty():
143                 if immediateExitRequested:
144                     break
145                     time.sleep(5)
146             print("RabbitMQ queue is empty. Continuing...")
147
148             break
149     except mysql.connector.Error as err:
150         print(f"MySQL error (status DB) for UUID {uuid}: {err}")
151
152         time.sleep(statusPollInterval)
153     try:
154         conn1 = mysql.connector.connect(**db1Config)
155         cursor1 = conn1.cursor()
156         cursor1.execute("SELECT totalDuration FROM mapTimes " \
157                         "WHERE mapID = %s LIMIT 1", (uuid,))
158         result = cursor1.fetchone()
159         dbValue = result[0] if result else ""
160         cursor1.close()
161         conn1.close()
162     except mysql.connector.Error as err:
163         print(f"MySQL error (main DB) for UUID {uuid}: {err}")
164         dbValue = ""
165
166     if immediateExitRequested:
167         updatedRows.append(row)
168         break
```

Abbildung 4.96.: messen.py Main 3

```
170     while len(row) < len(headers):
171         row.append("")
172         row[headers.index(uuidColumnName)] = uuid
173         row[headers.index(dbValueColumnName)] = dbValue
174
175         updatedRows.append(row)
176
177     for remainingRow in reader:
178         updatedRows.append(remainingRow)
179
180 with open(csvPath, mode="w", newline="") as file:
181     writer = csv.writer(file)
182     writer.writerow(headers)
183     writer.writerows(updatedRows)
184
185 print("CSV fully processed.")
```

Abbildung 4.97.: messen.py Main 4

```
33     def isQueueEmpty():
34         try:
35             response = requests.get(rabbitmqApiUrl, auth=(rabbitmqUser,
36                                         rabbitmqPassword))
37             if response.ok:
38                 data = response.json()
39                 messages = data.get("messages", -1)
40                 print(f"Queue '{rabbitmqQueue}' has {messages} messages.")
41                 return messages == 0
42             else:
43                 print(f"Failed to get queue status: \
44                      {response.status_code} {response.text}")
45                 return False
46         except Exception as e:
47             print(f"Error checking RabbitMQ queue: {e}")
48             return False
```

Abbildung 4.98.: messen.py isQueueEmpty()

```
61  def watchForExit():
62      global exitRequested
63      print("Press 'X' to stop the script gracefully.")
64      keyboard.wait("x")
65      print("\nExit requested. Finishing current row and saving...")
66      exitRequested = True
67
68  def watchForImmediateExit():
69      global immediateExitRequested
70      print("Press 'Q' to stop the script gracefully immediately.")
71      keyboard.wait("q")
72      print("\nExit requested. saving ...")
73      immediateExitRequested = True
```

Abbildung 4.99.: messen.py Exit Funktion

4.7.3. messung.sh

Das *messung.sh*-Skript wird dazu verwendet, verschiedene Durchläufe des Skripts *messen.py* (siehe 4.7.2) zu automatisieren. Es verfügt über eine Liste von Werten für die Worker-Anzahl (Abbildung 4.100), welche nacheinander als Argument für *messen.py* übergeben werden. Bevor eine Messreihe gestartet wird, skaliert das Skript automatisch über *kubectl* die benötigte Anzahl an Workern (Abbildung 4.101). Dabei wird geprüft, ob alle benötigten Pods laufen, bevor das Skript weiterarbeitet (Abbildung 4.102). Da *messen.py* mit Administratorrechten gestartet werden muss, aber *kubectl* nicht mit *sudo* aufgerufen werden darf, verfügt das Skript über einen Mechanismus, bei dem der Nutzer einmalig zu Beginn das Admin-Passwort eingeben muss.

```

1  #!/bin/bash
2
3  workerCounts=(1 2 4 8 9 16 18 25 32 36 49 50 64 72 98 100 128 144 196 256)
4
5  deploymentName="wfcworker-deployment"
6  namespace="default"
7  podLabelSelector="app=wfcworker"
8
9  read -s -p "Bitte gib dein sudo-Passwort ein: " sudoPassword
10 echo

```

Abbildung 4.100.: messung.sh Main 1

```

38  for X in "${workerCounts[@]}"; do
39      echo "==> Skaliere $deploymentName auf $X Worker..."
40      kubectl scale deployment "$deploymentName" --replicas="$X" -n "$namespace"
41
42      waitForPodsRunning "$X"
43
44      echo "==> Führe messen.py mit X=$X aus..."
45      runWithSudo python3 messen.py "$X"
46
47      echo "==> Durchlauf für X=$X abgeschlossen."
48      echo
49  done
50
51  echo "Alle Durchläufe abgeschlossen."

```

Abbildung 4.101.: messung.sh Main 2

```
16  waitForPodsRunning(){
17      local expectedCount=$1
18      echo "Warte, bis $expectedCount Pods mit Status 'Running' aktiv sind ..."
19
20      while true; do
21          runningCount=$(kubectl get pods -n "$namespace" -l "$podLabelSelector" \
22              --field-selector=status.phase=Running \
23              --no-headers 2>/dev/null | wc -l)
24
25          if [[ "$runningCount" -eq "$expectedCount" ]]; then
26              echo "Alle $expectedCount Pods sind 'Running'."
27              break
28          fi
29
30          echo "Aktuell '$runningCount' von '$expectedCount' Pods 'Running' - warte 5"
31          sleep 5
32      done
33
34      echo "Warte zusätzlich 60 Sekunden für Broker-Registrierung..."
35      sleep 60
36  }
```

Abbildung 4.102.: messung.sh Funktionen

4.7.4. Graphing Tool

Das Graphing Tool dient zur Visualisierung der Messergebnisse und benutzt hierfür *Matplotlib* und *Scipy* (für Polynom-Regression) - (siehe Abbildung 4.103). Da die Daten vierdimensional sind, wird jeweils ein Wert für die x- und y-Achse bestimmt. Danach wird für jede Kombination aus den Werten der übrigen Dimensionen ein Graph gezeichnet. Da die gemessene Zeit abhängig von der Kantenlänge der Map ermittelt wird, ist es eine sinnvolle Darstellungsweise, die Kantenlänge auf die x-Achse und die gemessene Zeit auf die y-Achse zu legen. So zeigt jeder Graph eine bestimmte Kombination aus der Anzahl der Worker-Services und der Anzahl der Abschnitte, in die die jeweilige Map in dieser Messung unterteilt wurde (siehe Abbildung 4.107 bis 4.110). Zusätzlich wird für jeden Graphen mit einer Polynom-Regression ein *Fit* berechnet. Mit diesen Werten kann zukünftig für Berechnungen eine geschätzte Berechnungsdauer bestimmt werden (siehe Abbildung 4.105 und 4.106). Außerdem unterstützt das Skript mehrere Optionen sowie die Möglichkeit, entweder einen spezifischen Messdurchgang oder den Durchschnitt zu verwenden (da jede Messung 3 mal durchgeführt wurde). Des Weiteren kann die Anzahl der Graphen mit Start- und Endpunkt für Übersichtlichkeit eingestellt werden und es gibt die Möglichkeit, statt Kurven nur die Datenpunkte darzustellen (siehe Abbildung 4.104).

```
1 import pandas as pd
2 import matplotlib.pyplot as plt # type: ignore
3 import argparse
4 import sys
5 import numpy as np
6 from scipy.optimize import curve_fit # type: ignore
```

Abbildung 4.103.: Graphing Imports

```
149 if __name__ == "__main__":
150     parser = argparse.ArgumentParser(description="Plot experimental data from a CSV")
151     parser.add_argument("csv_file", help="Path to the CSV file")
152     parser.add_argument("--start", type=int, default=0,
153                         help="Start index of groups to plot")
154     parser.add_argument("--end", type=int, default=None,
155                         help="End index of groups to plot")
156     parser.add_argument("--points-only", action="store_true",
157                         help="Plot only points (no lines)")
158     parser.add_argument("--runs", nargs="+", help="Specify which Run columns " \
159                         "to include (e.g. run1 Run2 RUN3)")
160     parser.add_argument("--poly-degree", type=int, default=2,
161                         help="Degree of polynomial regression (if used)")
162     parser.add_argument("--average-runs", action="store_true",
163                         help="Plot the average of all selected runs instead of each")
164     parser.add_argument("--regression-type", choices=["poly", "logistic"],
165                         default="poly",
166                         help="Type of regression to fit: 'poly' or 'logistic' " \
167                         "(default: poly)")
168
169     args = parser.parse_args()
170
171     plotExperimentData(
172         csvFile=args.csv_file,
173         startIndex=args.start,
174         endIndex=args.end,
175         pointsOnly=args.points_only,
176         runColumns=args.runs,
177         polyDegree=args.poly_degree,
178         averageRuns=args.average_runs,
179         regressionType=args.regression_type
180     )
```

Abbildung 4.104.: Graphing Main

```

112     def fitAndPlotRegression(xClean, yClean, regressionType, polyDegree, label):
113         try:
114             if regressionType == "poly":
115                 coeffs = np.polyfit(xClean, yClean, deg=polyDegree)
116                 fitFn = np.poly1d(coeffs)
117                 yFit = fitFn(xClean)
118
119                 ssRes = np.sum((yClean-yFit)**2)
120                 ssTot = np.sum((yClean-np.mean(yClean))**2)
121                 rSquared = 1 - ssRes/ssTot if ssTot != 0 else 0
122
123                 xSorted = np.sort(xClean)
124                 ySortedFit = fitFn(xSorted)
125                 plt.plot(xSorted, ySortedFit, linestyle="--", linewidth=1.5,
126                           alpha=0.7,
127                           label=f"Fit: {label} (poly, R²={rSquared:.7f})")

```

Abbildung 4.105.: Graphing findAndPlotRegression() 1

```

129     equationTerms = []
130     for j, coef in enumerate(coeffs[::-1]):
131         power = j
132         if abs(coef) < 1e-6:
133             continue
134         if power == 0:
135             equationTerms.append(f"{coef:.7f}")
136         elif power == 1:
137             equationTerms.append(f"{coef:.7f}x")
138         else:
139             equationTerms.append(f"{coef:.7f}x^{power}")
140     equation = " + ".join(equationTerms).replace("+ -", "- ")
141
142     textX = xSorted[-1]
143     textY = ySortedFit[-1]
144     plt.text(textX, textY, f"$y = {equation}$", fontsize=8, ha="right",
145               va="bottom", alpha=0.7)
146 except Exception as e:
147     print(f"Failed to fit {regressionType} regression for {label}: {e}")

```

Abbildung 4.106.: Graphing findAndPlotRegression() 2

```
10  def plotExperimentData(csvFile, startIndex=0, endIndex=None, pointsOnly=False,
11  |   |   |   |   |   |   runColumns=None, polyDegree=2, averageRuns=False,
12  |   |   |   |   |   |   regressionType="poly"):
13  # LOAD CSV-FILE
14  df = pd.read_csv(csvFile)
15
16  colMap = {col.lower(): col for col in df.columns}
17  df.columns = [col.lower() for col in df.columns]
18
19  required = ["size", "parts", "worker"]
20  for col in required:
21      if col not in df.columns:
22          sys.exit(f"Missing required column: {col}")
23
24  if runColumns is None:
25      runColumns = [col for col in df.columns if col.startswith("run")]
26  else:
27      runColumns = [col.lower() for col in runColumns]
28      for runCol in runColumns:
29          if runCol not in df.columns:
30              sys.exit(f"Run column {runCol} not found in CSV-file.")
31
32  df["size"] = pd.to_numeric(df["size"], errors="coerce")
33  for runCol in runColumns:
34      df[runCol] = pd.to_numeric(df[runCol], errors="coerce")
35
36  df = df.dropna(subset=["size"])
37
38  # grouping by parts and worker
39  groups = list(df.groupby(["parts", "worker"]))
```

Abbildung 4.107.: Graphing plotExperimentData() 1

```
41     if endIndex is None:
42         endIndex = len(groups)
43     selectedGroups = groups[startIndex:endIndex]
44
45     plt.figure(figsize=(12,8))
46
47     for (parts,worker), group in selectedGroups:
48         labelBase = f"{parts} - {worker}"
49
50         if averageRuns:
51             x=group["size"].values
52             runData=group[runColumns].values
53             y=np.nanmean(runData, axis=1)
54
55             mask=~np.isnan(x) & ~np.isnan(y)
56             xClean=x[mask]
57             yClean=y[mask]/1000
58
59             if len(xClean)<2:
60                 continue
61
62             label=f"{labelBase}-avg({', '.join(runColumns)})"
63             marker="o"
64
65             if pointsOnly:
66                 plt.scatter(xClean, yClean, label=label, marker=marker)
67             else:
68                 plt.plot(xClean, yClean, label=label, linestyle="--", marker=marker)
69
70             fitAndPlotRegression(xClean, yClean, regressionType, polyDegree, label)
```

Abbildung 4.108.: Graphing plotExperimentData() 2

```

71     else:
72         for i, runCol in enumerate(runColumns):
73             x=group["size"].values
74             y=group[runCol].values
75
76             mask=~np.isnan(x) & ~np.isnan(y)
77             xClean=x[mask]
78             yClean=y[mask]
79
80             if len(xClean)<2:
81                 continue
82
83             originalLabel = colMap.get(runCol, runCol)
84             label = f"{labelBase} - {originalLabel}"
85             marker = "o" if i % 2 == 0 else "x"
86             linestyle = "-" if not pointsOnly else "None"
87
88             if pointsOnly:
89                 plt.scatter(xClean, yClean, label=label, marker=marker)
90             else:
91                 plt.plot(xClean, yClean, label=label, linestyle=linestyle,
92                           marker=marker)
93
94             fitAndPlotRegression(xClean, yClean, regressionType, polyDegree,
95                                   label)

```

Abbildung 4.109.: Graphing plotExperimentData() 3

```

97     plt.xlabel("Size")
98     plt.ylabel("Measurement")
99     plt.title("Experiment Results by Parts & Worker")
100    plt.legend(
101        loc="center left",
102        bbox_to_anchor=(1.0, 0.5),
103        fontsize="small",
104        title="Legend"
105    )
106    plt.grid(True)
107    #plt.yscale("log")
108    plt.tight_layout(rect=[0,0,0.85,1])
109    plt.show()

```

Abbildung 4.110.: Graphing plotExperimentData() 4

4.7.5. build.sh

Das *build*-Skript führt hintereinander alle Befehle aus, welche zum Erstellen der einzelnen im Cluster verwendeten Docker-Images benötigt werden (siehe Abbildung 4.111).

```
1  #!/bin/bash
2  sudo docker build -t imgtimedb ./services/timedb
3  sudo docker build -t imgtimekeeper ./services/timekeeper
4  sudo docker build -t imgdb ./services/db
5  sudo docker build -t imgmanager ./services/manager
6  sudo docker build -t imgdistributor ./services/distributor
7  sudo docker build -t imghub ./services/hub
8  sudo docker build -t imgworker ./services/worker
```

Abbildung 4.111.: build.sh Skript

4.7.6. push.sh

Das *push*-Skript versieht alle im Cluster verwendeten Docker-Images mit einem *tag*, welcher die Adresse sowie die Versionsnummer beinhaltet. Anschließend werden die Images in den Docker-Hub hochgeladen. Adresse und Version können zu Beginn des Skripts als Variable festgelegt werden (siehe 4.112).

4.7.7. rebirth.sh

Das *rebirth*-Skript stoppt und löscht sämtliche Container. Anschließend wird zusätzlich das Deployment heruntergefahren und eine Löschung aller Pods erzwungen. Dies geschieht, um sicherzustellen, dass keine blockierten Prozesse vorhanden bleiben. Danach werden alle vorhandenen Docker-Images gelöscht, neu erstellt und in den Docker-Hub geladen, indem die Skripte *build.sh* und *push.sh* ausgeführt werden. Im letzten Schritt wird das Deployment mit den neu erstellten Images wieder gestartet (Abbildung 4.113).

4.7.8. rebuild.sh

Das *rebuild*-Skript erwartet als Argument einen Servicenamen. Das Skript löscht dann das zu dem Namen gehörige Docker-Image, erstellt es neu und lädt es in den Docker-Hub (Abbildung 4.114).

```
1  #!/bin/bash
2  ADDR=pfropfen
3  VER=latest
4
5  sudo docker tag imgmanager $ADDR/imgmanager:$VER
6  sudo docker tag imgdistributor $ADDR/imgdistributor:$VER
7  sudo docker tag imghub $ADDR/imghub:$VER
8  sudo docker tag imgworker $ADDR/imgworker:$VER
9  sudo docker tag imgdb $ADDR/imgdb:$VER
10 sudo docker tag imgtimedb $ADDR/imgtimedb:$VER
11 sudo docker tag imgtimekeeper $ADDR/imgtimekeeper:$VER
12
13 sudo docker push $ADDR/imgmanager:$VER
14 sudo docker push $ADDR/imgdistributor:$VER
15 sudo docker push $ADDR/imghub:$VER
16 sudo docker push $ADDR/imgworker:$VER
17 sudo docker push $ADDR/imgdb:$VER
18 sudo docker push $ADDR/imgtimedb:$VER
19 sudo docker push $ADDR/imgtimekeeper:$VER
```

Abbildung 4.112.: push.sh Skript

```
1  #!/bin/bash
2  sudo docker container stop $(sudo docker ps -a)
3  |   | && sudo docker container rm $(sudo docker ps -a)
4  kubectl delete -f wfcdeploy.yaml
5  kubectl delete --all pods --grace-period=0 --force
6  sudo docker image rm -f $(sudo docker image ls)
7  bash build.sh
8  bash push.sh
9  kubectl apply -f wfcdeploy.yaml
```

Abbildung 4.113.: rebirth.sh Skript

```
1  #!/bin/bash
2  ADDR=pfropfen
3  VER=latest
4  SERVICE=$1
5  IMAGE="img$1"
6
7  sudo docker image rm $IMAGE
8  sudo docker build -t $IMAGE ./services/$SERVICE
9  sudo docker tag $IMAGE $ADDR/$IMAGE:$VER
10 sudo docker push $ADDR/$IMAGE:$VER
```

Abbildung 4.114.: rebuild.sh Skript

4.7.9. bashrc

Bashrc ist eine Linux-eigene Datei, in der sogenannte *Aliases* für Befehle eingetragen werden können. Auf diesem Weg ist es möglich, innerhalb des Terminals durch kurze, selbst erstellte Befehle komplexere Eingaben zu ermöglichen. Dies bietet sich vor allem bei langen und oft verwendeten Befehlen an. So können beispielsweise mit dem Befehl *killall* sämtliche Container zunächst gestoppt und anschließend gelöscht werden (siehe Abbildung 4.115).

```
5 alias brc='nano ~/.bashrc'
6 alias psa='sudo docker ps -a'
7 alias killall='sudo docker container stop $(sudo docker ps -a -q) \
8 | | | | && sudo docker container rm $(sudo docker ps -a -q)'
9 alias killimg='sudo docker image rm -f $(sudo docker image ls)'
10 alias logs='sudo docker logs'
11 alias slogs='sudo docker service logs'
12 alias startreg='sudo docker run -d -p 9000:5000 --restart always \
13 | | | --name registry registry:2'
14 alias rebirth='bash rebirth.sh'
```

Abbildung 4.115.: bashrc Datei

4.8. Problemlösungen

Im folgenden Abschnitt werden spezielle Probleme sowie deren Lösungen behandelt, welche im Verlauf des Projekts entstanden sind. Generell waren die Werke von Hightower (Hightower 2022), Hausenblas (Hausenblas 2019) und Baier (Baier 2017) in Bezug auf Kubernetes, sowie das Git-Repository von Maxim Gumin (Gumin 2025) in Bezug auf den WFC-Algorithmus eine große Hilfe, um Verständnis für die Thematik aufzubauen.

4.8.1. Broker vs. Message-Queue

Am Anfang des Projekts wurde versucht, die Kommunikation zwischen den Services mit Hilfe des Message-Brokers *Kafka* zu realisieren. Allerdings arbeitet Kafka mit einem Speichermodell, das dazu führt, dass Nachrichten, welche bereits durch einen Consumer abgeholt wurden, weiterhin im Speicher verfügbar sind. Das führte dazu, dass ein bestimmter Chunk von mehreren Workern abgerufen werden konnte. Für dieses Projekt bot sich daher die Verwendung einer klassischen Message-Queue wie RabbitMQ an, in der Nachrichten nach Abruf aus dem Speicher gelöscht werden.

4.8.2. Dockerfile Layer

Während der Entwicklung mussten sehr häufig Docker-Images neu erstellt werden. Die Bauweise von Docker-Images funktioniert schichtweise, dies ist eines der Grundprinzipien von Docker und erlaubt die effiziente Speicherung und Übertragung von Docker-Images. Allerdings ist hierbei die Reihenfolge der verschiedenen Schichten wichtig. Eine generelle Bauweise eines Images erfolgt in der Reihenfolge: Grundlegende Image, Dateien kopieren, benötigte Befehle ausführen, Einstiegspunkt einrichten. Doch hat sich in diesem Projekt dabei ein Problem ergeben, da der dritte Schritt, die benötigten Befehle, deutlich mehr Zeit in Anspruch nimmt als die vorherigen Schritte. So hat es sich hier gelohnt, die zweite und dritte Schicht zu tauschen. Dadurch konnte eine viel bessere Performance beim Erstellen von Images erreicht werden, da die Installationsbefehle nicht für jeden Build erneut ausgeführt werden mussten.

4.8.3. Robustheit RabbitMQ

Sehr früh während des Projektes wurde sich für die RabbitMQ Message-Queue und deren zugehörige Python-Bibliothek entschieden. Die Implementierung lief lange Zeit ohne größere Probleme und wurde immer weiter ausgebaut. Doch bei den späteren Messungen ergaben sich Probleme aus dem Grund, dass eine Blocking-Connection benutzt wurde. RabbitMQ verwendet für Consumer ein Heartbeat-Protokoll, welches in regelmäßigen Abständen bei den Consumern nachfragt, ob diese noch erreichbar sind. Für den Fall, dass ein Consumer zwei Mal hintereinander eine Heartbeat-Anfrage nicht beantwortet, wird die Verbindung abgebrochen und die unbestätigte Nachricht wird wieder zur Verteilung freigegeben. Hier entsteht nun das Problem, da der Consumer während der Bearbeitung einer Anfrage nicht mehr auf Heartbeat-Anfragen antworten kann. Dementsprechend können Berechnungen, welche länger dauern als zwei Heartbeat-Intervalle, nicht bearbeitet werden. Besonders bei unverteilten Berechnungen von Maps kann dies sehr schnell der Fall sein. Die beste Lösung hierfür wäre es, die Blocking-Connection aufzugeben und stattdessen eine asynchrone Herangehensweise zu wählen. Dies würde jedoch erfordern, das komplette Consumer-Skript von Grund auf neu zu entwickeln. Auf Grund dessen, dass sich das Projekt jedoch schon in der Evaluierungs-Phase befand, wurde sich stattdessen dazu entschieden, das Heartbeat-Intervall auf einen sehr hohen Wert zu setzen und diesen dadurch effektiv zu deaktivieren. Dies schränkt jedoch leicht die Robustheit des RabbitMQ-Services ein und geht davon aus, dass die Consumer keine unvorhergesehenen fatalen Abstürze erleiden.

4.8.4. Robustheit Worker

Einhergehend mit dem Problem der RabbitMQ Heartbeat-Robustheit (siehe 4.8.3) wurde auch der Worker-Service negativ beeinflusst. Im Fall, dass die abgearbeitete Nachricht nicht bestätigt werden kann, weil der RabbitMQ-Service die Verbindung geschlossen hat, erleidet dieser einen fatalen Fehler und stürzt ab. Jedoch hat der Worker-Service zu diesem Zeitpunkt schon die Berechnung vollständig erledigt und die entsprechenden Daten in der Datenbank gespeichert. Dementsprechend kann die Messung der Zeit hier nicht mehr verwendet werden, obwohl die eigentlichen Daten in einem praktischen Fall noch vorhanden waren. Dies führt jedoch zu einem Problem, da die Nachricht aus Sicht des RabbitMQ-Services nicht bearbeitet wurde und diese darauffolgend an einen anderen oder denselben Worker erneut verteilt wird. Dies führt nun zu einem Kreislauf, bei dem der Worker nun wieder abstürzt, weil die Nachricht nicht bestätigt werden konnte. Dies stellt aufgrund der Heartbeat-Änderung kein großes Problem mehr dar. Trotzdem wurde eine Try-Except-Lösung im Worker-Service implementiert, um zu gewährleisten, dass der Service theoretisch nicht-problematische Nachrichten, trotz einer vorher erhaltenen problematischen Nachricht, weiter abarbeiten kann.

4.8.5. Messungen per Skript

Bei einer automatisierten Messreihe durch ein Skript (siehe Abschnitt 4.7.2 und 4.7.3) werden nacheinander eine Reihe von Maps generiert. Bei der Ausführung von langen Messreihen wurde festgestellt, dass es vorkommen kann, dass bereits die nächste Generierung gestartet wird, obwohl sich noch ein Ticket im Kanal der Message-Queue befindet (siehe auch Abschnitt 4.8.6). Auch wenn RabbitMQ nach dem FIFO-Prinzip arbeitet und dies somit grundsätzlich kein Problem darstellt, kann es durch die hohe Menge an Tickets bei Messungen zu Verfälschungen der Messzeiten für die Evaluation führen. Daher wurde zusätzlich ein Mechanismus eingebaut, welcher immer vor dem Start einer neuen Messung prüft, ob die Message-Queue noch Nachrichten enthält.

4.8.6. Invalide Messungen

Während der Evaluation wurde festgestellt, dass sämtliche bis zu dem Zeitpunkt gemessenen Zeiten nicht valide waren. Der Grund war ein Fehler im Skript, mit dem automatisierte Messungen durchgeführt werden. Dabei wurde die Boolean-Variable *computed* eines Eintrags der Datenbank mit der entsprechenden Map-ID auf den Wert *True* geprüft, um festzustellen, ob eine Map vollständig berechnet wurde. Allerdings bezieht sich diese Variable auf die Berechnung eines einzelnen Chunks und nicht

auf eine gesamte Map. Es muss daher für alle einer Map-ID zugehörigen Chunks geprüft werden, ob diese *computed* sind. Nachdem der Fehler behoben wurde, mussten sämtliche Messungen wiederholt werden.

4.8.7. Entropy Tolerance

Nach einigem Experimentieren stellte sich heraus, dass Maps, zumindest mit geringer bis mittlerer Größe, bessere Ergebnisse liefern, wenn eine Entropietoleranz von 5 verwendet wird. Dies geht jedoch mit den in Kapitel 3.2.5 erwähnten Problemen einher. Doch in der Praxis hat sich kein Fall ergeben, bei dem eine fehlgeschlagene Map generiert wurde. Um handfestere Beweise für die vermuteten Probleme zu erlangen, wurde manuell eine Situation für den WFC-Algorithmus erstellt, in der es schnell zu einer fehlgeschlagenen Map kommen kann. Nach mehreren Tests bewies sich nun auch praktisch, dass eine Entropietoleranz von 0 die einzige Möglichkeit ist, sicherzustellen, dass keine fehlgeschlagene Map generiert werden kann. Nachdem dieses Problem so deutlicher hervorgehoben werden konnte, wurde sich dafür entschieden, die Messungen für die Evaluation mit einer Entropietoleranz von 0 durchzuführen. Dies geht zusätzlich auch mit besserer Performance einher, da für jeden von 0 abweichenden Wert der Entropietoleranz eigene Listen mit Potenzialen erstellt und verwaltet werden müssen (siehe auch Abschnitt 4.2).

5. Evaluation

5.1. Methodik und Aufbau

Wie in Kapitel 3 (3.5) beschrieben, werden für die Evaluation eine Reihe von Messungen durchgeführt. Gemessen werden sowohl die Zeiten zum Generieren einer Map als auch die einzelnen Zeiten zum Berechnen der Chunks durch die Worker-Einheiten. Die Messreihe beinhaltet 3 Variablen:

- **Size:** Kantenlänge der Map in Anzahl der Tiles
- **Parts:** Anzahl Abschnitte, in die eine Map zerlegt wird
- **Worker:** Anzahl der zu verwendenden Worker

Die Messreihe wird systematisch mit verschiedenen Werten durchgeführt. Die Kantenlänge beginnt bei 16 und inkrementiert in 16er-Schritten bis 256. Die Anzahl der Parts reicht von 1 (unverteilt) bis maximal 256 Abschnitte, abhängig von den durch die Map-Größe gegebenen Möglichkeiten (siehe Kapitel 3.3). Die Anzahl der Worker ist abhängig von der Anzahl der Abschnitte und ist, sofern möglich, immer ein Viertel, die Hälfte und die volle Anzahl der Parts. Abbildung 5.1 zeigt einen Ausschnitt der vorbereiteten Messtabelle. Die vollständige Tabelle kann im Git-Repository (B.1) eingesehen werden.

SIZE	PARTS	WORKER	p=2 ²	p k ²
16	1	1 OK	OK	
16	4	1 OK	OK	
16	4	2 OK	OK	
16	4	4 OK	OK	
16	16	4 OK	OK	
16	16	8 OK	OK	
16	16	16 OK	OK	
32	1	1 OK	OK	
32	4	1 OK	OK	
32	4	2 OK	OK	
32	4	4 OK	OK	
32	16	4 OK	OK	
32	16	8 OK	OK	
32	16	16 OK	OK	
32	64	16 OK	OK	
32	64	32 OK	OK	
32	64	64 OK	OK	
48	1	1 OK	OK	

Abbildung 5.1.: Beispielauszug der Datei messreihen.csv

5.1.1. Verwendete Hardware

Die Messungen werden auf dem zur Verfügung gestellten Cluster der TH Köln, Campus Gummersbach, durchgeführt, welches aus 9 Mac Pro Computern besteht. Die Hardware der Rechner ist je:

- Intel Xeon E5 (12 Kerne a 2,7 GHz)
- 64 GB 1866 MHz DDR3 Ram
- AMD FirePro D700 6 GB
- 500 GB HDD

Acht der verwendeten Computer werden für den Betrieb des Kubernetes-Clusters verwendet (Manager + 7 Worker-Nodes) und laufen mit einem Linux Ubuntu Server 22.04-Betriebssystem. Der neunte Computer läuft mit einem macOS Monterey 12.6.6-Betriebssystem und wird für Monitoring sowie Cluster-Interaktion verwendet (Abbildung 5.2).



Abbildung 5.2.: Spezifikation der Mac Pro Computer

5.2. Durchführung

Um die Messungen durchzuführen, wird ein speziell entwickeltes Skript ausgeführt (siehe Kapitel 4.7.2 und 4.7.3). Schritt für Schritt wird die vorgegebene Messreihe durchgeführt und die Ergebnisse als neuer Eintrag in die Messreihen-Datei geschrieben. Zusätzlich werden anschließend die gespeicherten Zeiten der Time-DB in CSV-Dateien exportiert (siehe auch Kapitel 4.3.8 und 4.7.1). Jede Messung wird genau dreimal wiederholt, um einen Mittelwert bilden zu können und Ausreißer zu identifizieren.

5.3. Messergebnisse

Innerhalb der Messreihen wurden inklusive aller Wiederholungen und Fehlmessungen 676 Maps mit insgesamt 50540 Chunks erzeugt. Der vollständige Inhalt der Datenbank kann im Git-Repository (siehe Anhang A.3) eingesehen werden. Die vollständige zusammengefasste Messtabelle, bestehend aus 618 (206x3) Messungen, kann im Anhang B.1 eingesehen werden. Die Abbildung 5.3 zeigt einen Ausschnitt der zusammengefassten Messtabelle.

Die Messtabelle setzt sich aus folgenden Parametern zusammen:

- **Row:** Nummer der Messung
- **SIZE:** Kantenlänge der Map in Tiles
- **PARTS:** Anzahl der Abschnitte der Map
- **WORKER:** Anzahl der beteiligten Worker
- **Run1:** Erste gemessene Zeit in Millisekunden
- **Run2:** Zweite gemessene Zeit in Millisekunden
- **Run3:** Dritte gemessene Zeit in Millisekunden

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
1	16	1	1	315	311	334
2	16	4	1	2001	1880	1737
3	16	4	2	972	998	867
4	16	4	4	519	401	494
5	16	16	4	3354	3270	3119
6	16	16	8	2891	2639	2952
7	16	16	16	2261	2220	2305
8	32	1	1	4951	2782	3869
9	32	4	1	2458	2306	2234
10	32	4	2	1322	1261	1165
11	32	4	4	491	484	675
12	32	16	4	3453	3428	3622
13	32	16	8	3229	2694	2841
14	32	16	16	2271	2020	2320
15	32	64	16	15252	12730	14913
16	32	64	32	12078	12446	11767
17	32	64	64	8464	8368	8772
18	48	1	1	20736	17865	19557
19	48	4	1	4368	3843	3806
20	48	4	2	1918	2116	2307

Abbildung 5.3.: Ausschnitt 1 der Messreihentabelle

5.4. Analyse/Auswertung

Im folgenden Abschnitt werden die Messergebnisse anhand von Graphen analysiert. Die Graphen wurden mit einem speziellen Skript erstellt (siehe Kapitel 4.7.4). Für einige der Graphen wurde dieses Skript leicht angepasst, um verschiedene Vergleiche zu ermöglichen. Jeder Graph beschreibt per Definition immer eine Funktionsvorschrift. Der Grad dieser Funktionsvorschrift gibt die höchste Potenz der Variablen an, welche in der Funktion vorkommt. Um einen möglichst präzisen Vergleich der Daten zu ermöglichen, wird bei der Erstellung eines Graphen ein sogenannter *fit* ermittelt, welcher möglichst der eigentlichen, aber nicht bekannten, Funktionsvorschrift entspricht. Die aufgetragenen Zeiten entsprechen immer dem aus den gemessenen Durchgängen gebildeten Durchschnittswert.

Abbildung 5.4 zeigt die gemessenen Zeiten von nicht parallelisierten (unverteilten) Maps abhängig von der Map-Größe bzw. der Kantenlänge. Auffällig dabei ist, dass die gemessenen Zeiten nicht linear, sondern polynomial zu der Kantenlänge steigen. Grundsätzlich ist dies auch zu erwarten, da die Map-Größe das Quadrat der Kantenlänge ist. Dennoch ist die Steigung größer, als man aufgrund dieser Tatsache erwarten würde. Abbildung 5.5 zeigt die selben Daten mit einer logarithmisch-skalierten y-Achse, wodurch diese Tatsache noch deutlicher wird. Dies hat vermutlich zwei Gründe. Der erste Grund ist, dass die Änderungen durch größere Maps weiter propagieren und somit mehr Rechenzeit in Anspruch nehmen. Der zweite Grund ist, dass für die Auswahl des nächsten zu kollabierenden Potenzials eine Schleife benötigt wird, welche über alle Potenziale iterieren muss. Diese Schleife wächst in der Anzahl der Berechnungen dementsprechend auch mit quadratischem Aufwand. Die Entropietoleranz könnte hier auch einen großen Unterschied machen; in dieser Messreihe wurden allerdings alle Messungen mit der Entropietoleranz von 0 durchgeführt (siehe auch Kapitel 3.2.5).

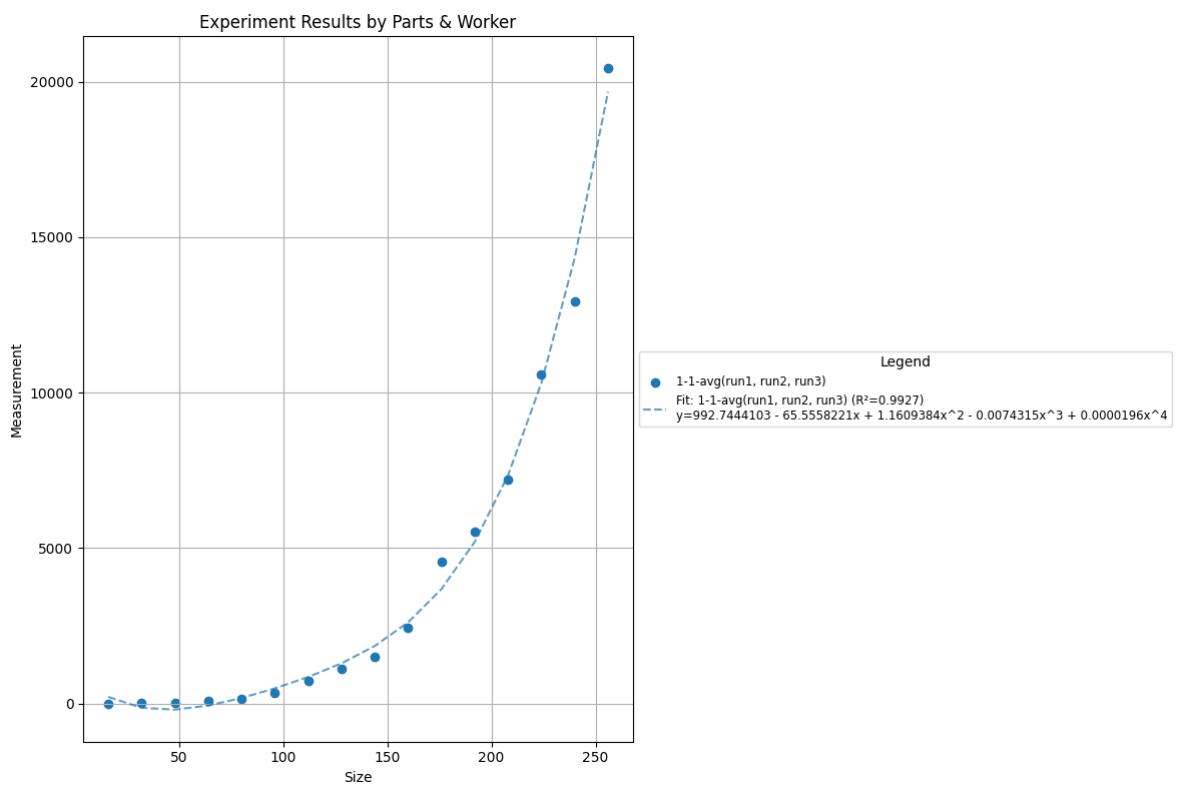


Abbildung 5.4.: Unverteilte Berechnung, x-Achse: Kantenlänge in Tiles, y-Achse: gemessene Zeit in Sekunden

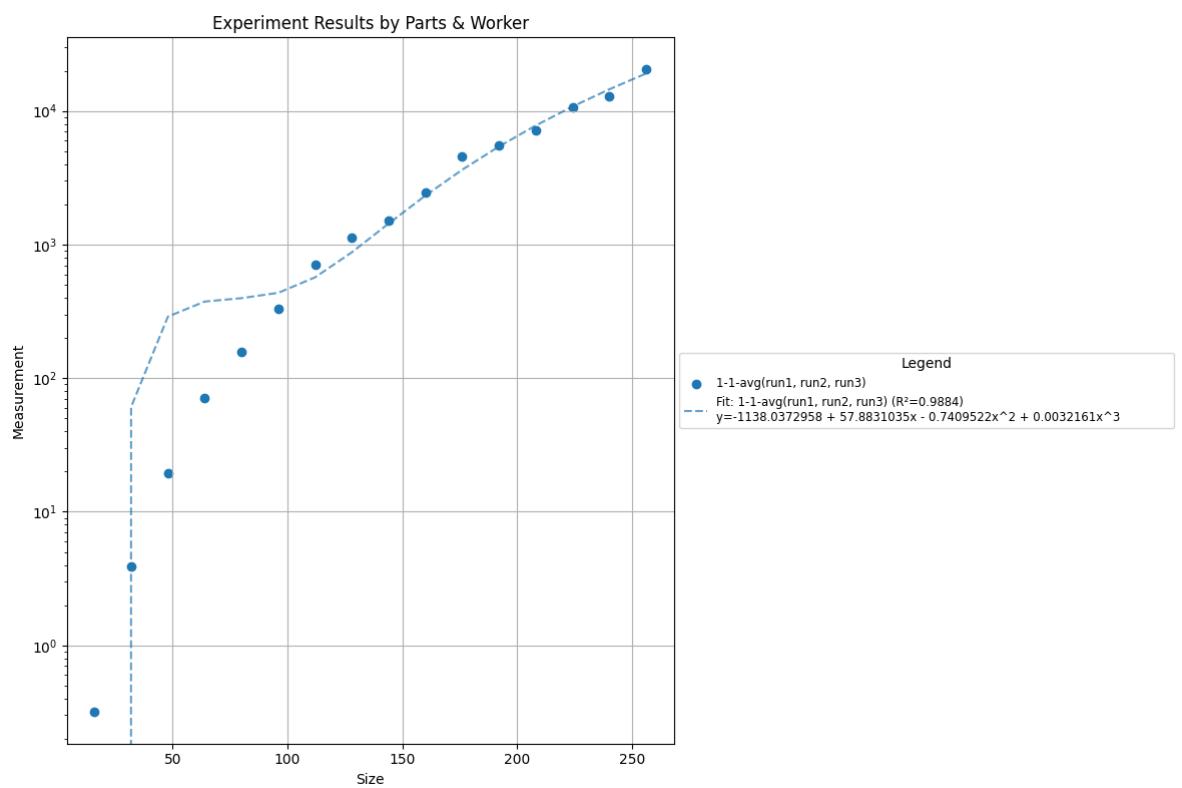


Abbildung 5.5.: Unverteilte Berechnung, logarithmische Darstellung, x-Achse: Kantenlänge in Tiles, y-Achse: gemessene Zeit in Sekunden

Die Abbildungen 5.6 - 5.9 zeigen die Graphen für die verschiedenen Verteilungen anhand der Anzahl der Parts. Die x-Achse beschreibt jeweils die Kantenlänge der generierten Maps in Tiles, die y-Achse zeigt die gemessene Zeit in Sekunden. Jeder Graph enthält 3 Kurven:

- **Grün:** Anzahl Worker = Die volle Anzahl der Parts
- **Orange:** Anzahl Worker = Die halbe Anzahl der Parts
- **Blau:** Anzahl Worker = Die viertel Anzahl der Parts

Auf Abbildung 5.6 und 5.7 sind die Graphen für die Verteilung von Maps in 4 und 16 Parts zu sehen. Diese Datenpunkte bilden in beiden Graphen eine relativ saubere Polynomfunktion ab. In beiden Graphen ist zu beobachten, dass eine Verteilung, bei der die Anzahl der Parts der Anzahl an Workern entspricht, die höchste Performance bzw. die geringste Rechenzeit erzielt. Die Performance scheint sich bei einer Verdopplung der Anzahl der Worker, wie zu erwarten, auch fast zu verdoppeln, doch geht etwas an Performance für das Networking verloren. Ein weiterer großer Unterschied zwischen den beiden Graphen ist, wie auf der y-Achse zu bemerken, dass die Aufteilung in 16 Parts statt 4, unabhängig von der relativen Anzahl der Worker, viel bessere Zeiten erzielt. Der Anstieg liegt bei ungefähr 20-mal schnellerer Rechenzeit.

Abbildung 5.8 zeigt den Graphen für die Verteilung von 64 Parts auf 16 bis 64 Worker. Im Gegensatz zu den in Abbildung 5.6 und 5.7 zu sehenden Graphen, welche eine polynomiale Steigung andeuten, sind diese Graphen nicht auf den ersten Blick einzuordnen. Die Graphen für die viertel und halbe Workeranzahl schwanken stark. Dies könnte jedoch mit der stark verbesserten Performance zu tun haben, da kleinere Abweichungen in individueller Chunk-Berechnungsdauer und Netzwerk-Delay größere Auswirkungen auf den zu beobachtenden Graph haben. Merkwürdig scheint jedoch, dass jeder Graph einen „Durchhänger“ an der fast gleichen Stelle hat. Möglicherweise lässt sich dies auf die Berechnungsdauer des Distributors zurückführen, denn dieser übernimmt bei kleineren Chunks einen größeren Rechenanteil. Dies könnte dazu führen, dass sich die Performance zuerst mit der steigenden Kantenlänge verringert und dann, nachdem die Chunkgröße groß genug ist, um den Distributor zu entlasten, zeitlich kurz ansteigt. Daraufhin fällt die Performance wieder ab, da dieser geringe Anstieg von der polynomial skalierenden Rechenanforderung eingeholt wird. Die Performance zwischen den verschiedenen Graphen verhält sich fast gleich zu den relativen Änderungen in der Worker-Anzahl. Eine weitere wichtige Erkenntnis aus diesem Graph ist, dass ohne Hardware-Limitierung der Algorithmus in der Lage ist, wie im untersten Graph zu sehen, in der Performance fast linear zu der Kantenlänge zu skalieren.

In Abbildung 5.9 sind die Graphen zu sehen, welche mit der Verteilung einer Map in 256 Parts einhergehen. Dieser scheint auf den ersten Blick etwas „chaotisch“, was möglicherweise an den, in Relation betrachtet, größeren Chunks liegt. Dies, zusammen

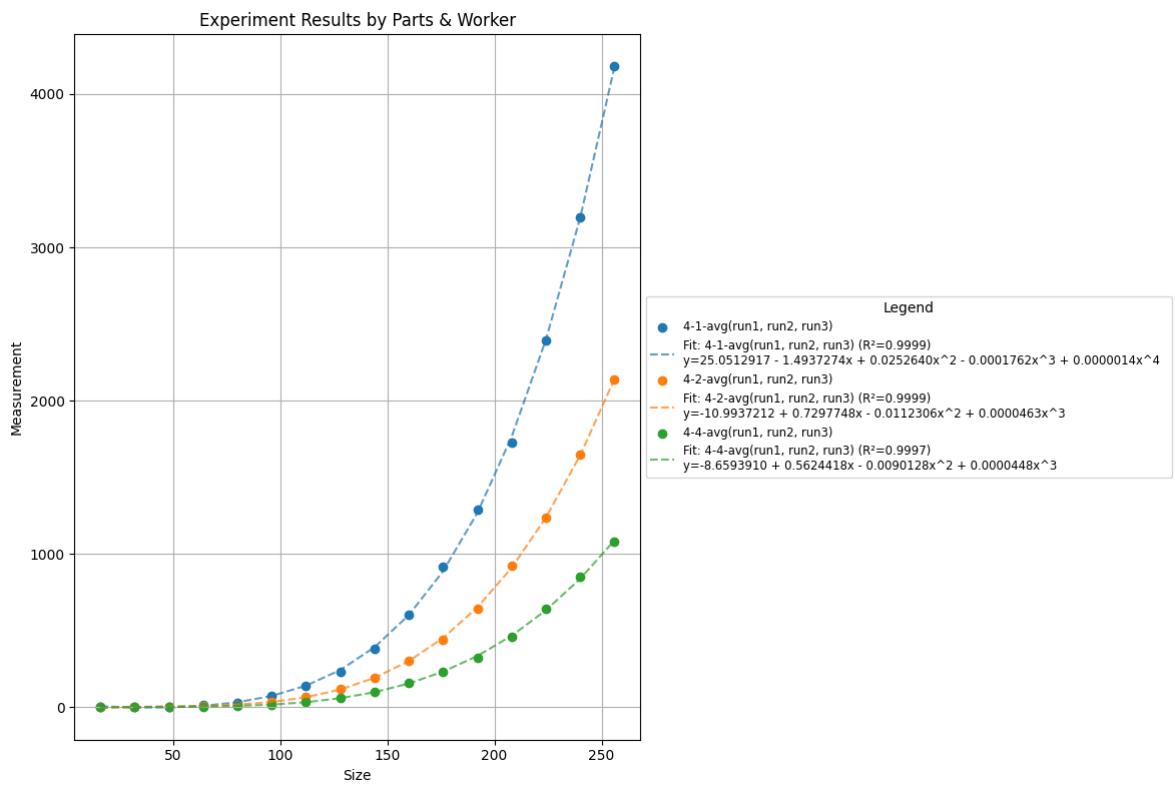


Abbildung 5.6.: Verteilung in 4 Parts

mit der immer noch guten Performance, ergibt ein auf den ersten Blick durchschnittliches Bild der ersten beiden Graphen. Hingegen verhält sich der Graph, bei dem die Anzahl der Parts der Anzahl der Worker entspricht, weiterhin stabil. Außer leichten Schwankungen, aufgrund des Distributor-Aufwands, verhält dieser sich weiterhin fast linear. Eine wichtige Anmerkung zu diesem Diagramm ist, dass die x-Achse bei Werten in der Nähe von 100 startet. Dies liegt daran, dass kleinere Maps nicht in 256 Teile geteilt werden können. Dieser Fakt muss somit in die Bewertung der Performance miteinbezogen werden.

Abbildung 5.10 zeigt die Performance basierend auf der Kantenlänge der Map, wobei jeder Graph eine Chunkgröße darstellt. Hier ist zu erkennen, dass größere Chunks generell eine bessere Performance liefern, doch ist die Performance bei relativ kleinen Maps schlechter. Dementsprechend gibt es einen „Break-even-Point“, bei dem die Performance eines kleineren Chunks von dem eines größeren überholt wird. Zum Beispiel ist hier zu sehen, dass bei einer Kantenlänge von ca. 110 die Performance

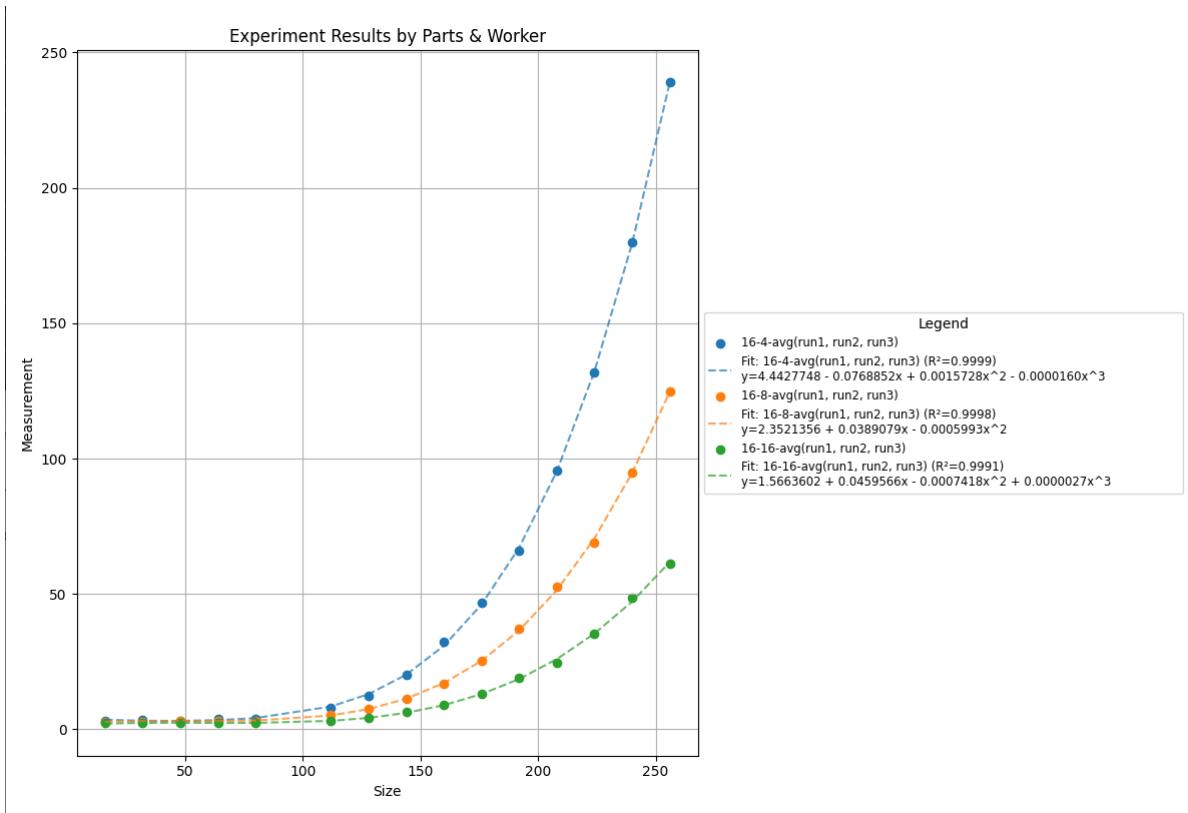


Abbildung 5.7.: Verteilung in 16 Parts

des Algorithmus mit einer Chunkgröße von 256 von dem mit 1024 überholt wird. Dementsprechend ist die korrekte Wahl der Chunkgröße abhängig von der vorhandenen Kantenlänge. In diesem Fall führt dies dazu, dass für Kantenlängen unter 110 eine Chunkgröße von 256 gewählt werden sollte und darüber die Chunkgröße 1024. Doch auch bei Werten unter 110 wird die Performance von 256 zuvor von einer kleineren Chunkgröße übertrumpft. Es ist stark anzunehmen, dass dieser Trend weitergeführt werden kann und die Performance der Chunkgröße bei steigender Kantenlänge sehr bald von einer noch größeren Chunkgröße überholt wird. Generell kann der Break-even-Point beliebig weit nach links oder rechts auf dem Graphen verschoben werden, unter der Bedingung, dass alle Chunkgrößen zur Wahl stehen, was in diesem Fall nicht möglich ist.

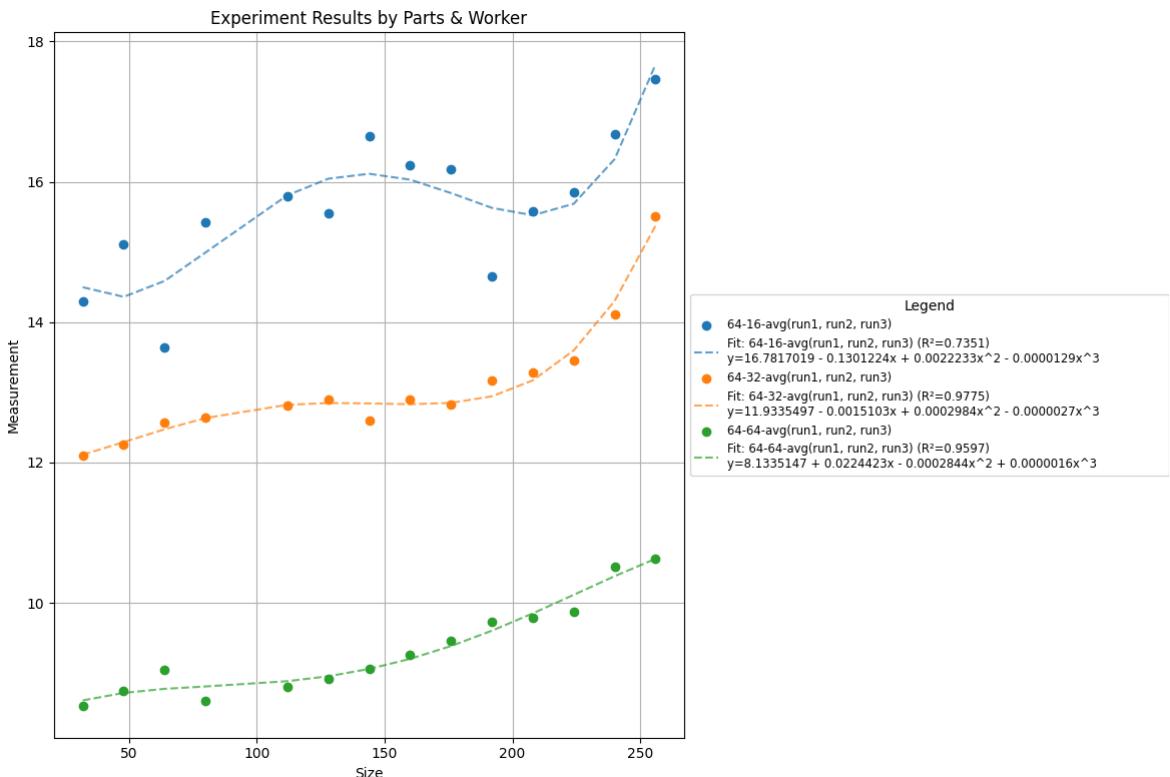


Abbildung 5.8.: Verteilung in 64 Parts

Abbildung 5.11 zeigt die Graphen zweier Funktionen, welche beide je 16 Worker für die Berechnung verwendet haben. Es ist zu erkennen, dass es auch hier einen Break-even-Point bei einer Kantenlänge von ca. 180 gibt. Dementsprechend ist es für den Fall, dass die benutzte Hardware 16 Worker laufen lassen kann, empfehlenswert, ab einer Kantenlänge von ca. 180 die Berechnung von einer Aufteilung von 16 (1 zu 1) auf die Verteilung von 64 (4 zu 1) zu wechseln. Dieser Fall, dass eine Verteilung in mehr Parts als die Anzahl der Worker ab dieser Stelle eine höhere Performanz erreicht, liegt daran, dass eine höhere Anzahl an Chunks besser mit steigender Kantenlänge skaliert. Im Gegensatz dazu wächst die Chunkgröße bei geringerer Verteilung schneller, in einem Verhältnis, welches dem invertierten Verhältnis zwischen den Anzahlen der Parts entspricht.

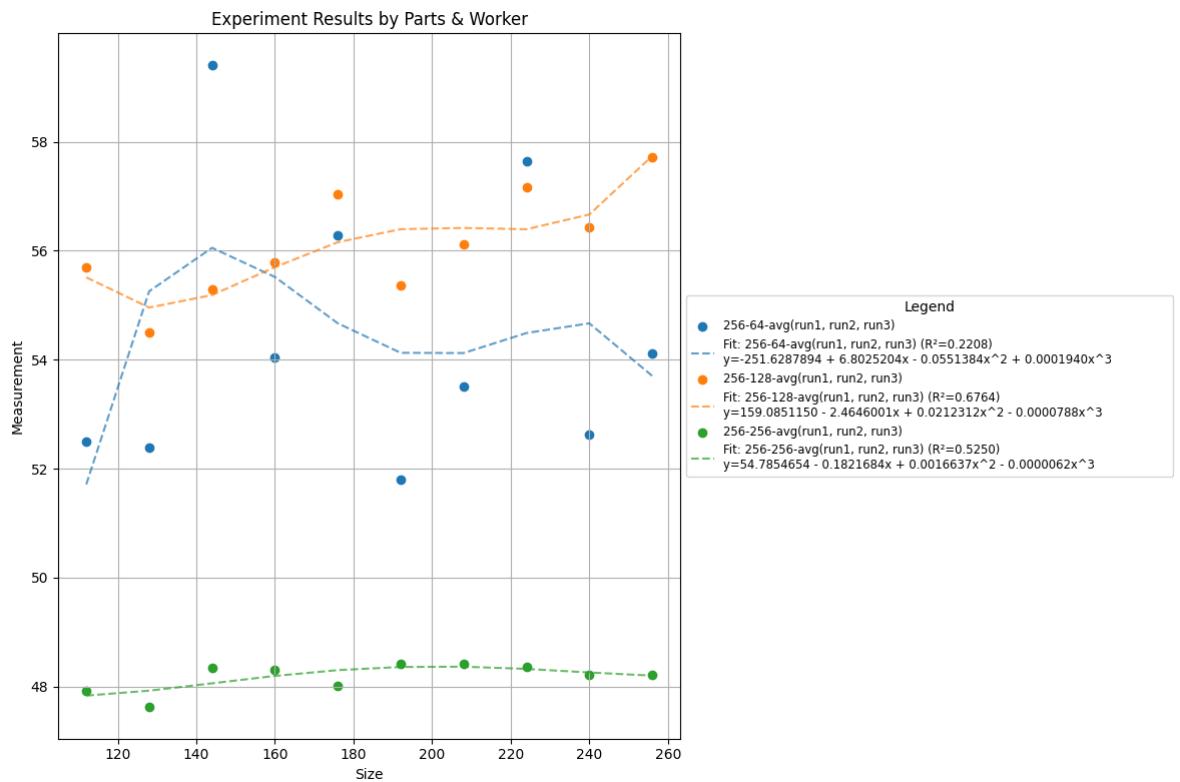


Abbildung 5.9.: Verteilung in 256 Parts

Es gilt:

$$\text{Chunkgröße} = \frac{\text{Kantenlänge}^2}{\text{Parts}}$$

Steigung der Chunkgröße (S) durch Ableitung:

$$S = \text{Kantenlänge} \cdot \frac{2}{\text{Parts}}$$

Daraus ergibt sich:

$$\frac{S_1}{S_2} = \frac{\text{Kantenlänge} \cdot \frac{2}{\text{Parts}_1}}{\text{Kantenlänge} \cdot \frac{2}{\text{Parts}_2}}$$

Nach Vereinfachung:

$$\frac{S_1}{S_2} = \frac{\text{Parts}_2}{\text{Parts}_1}$$

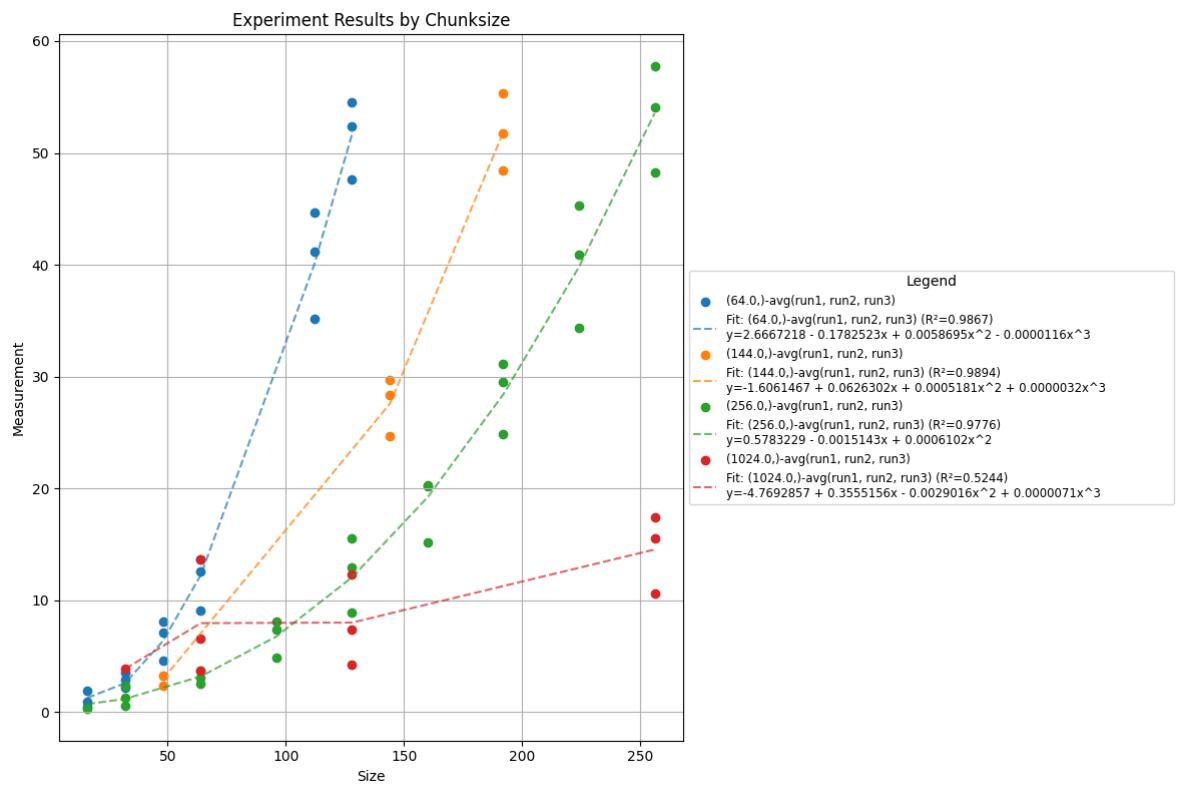


Abbildung 5.10.: Relation von Kantenlänge in Tiles zur gemessenen Zeit in Sekunden auf Basis der Chunkgröße

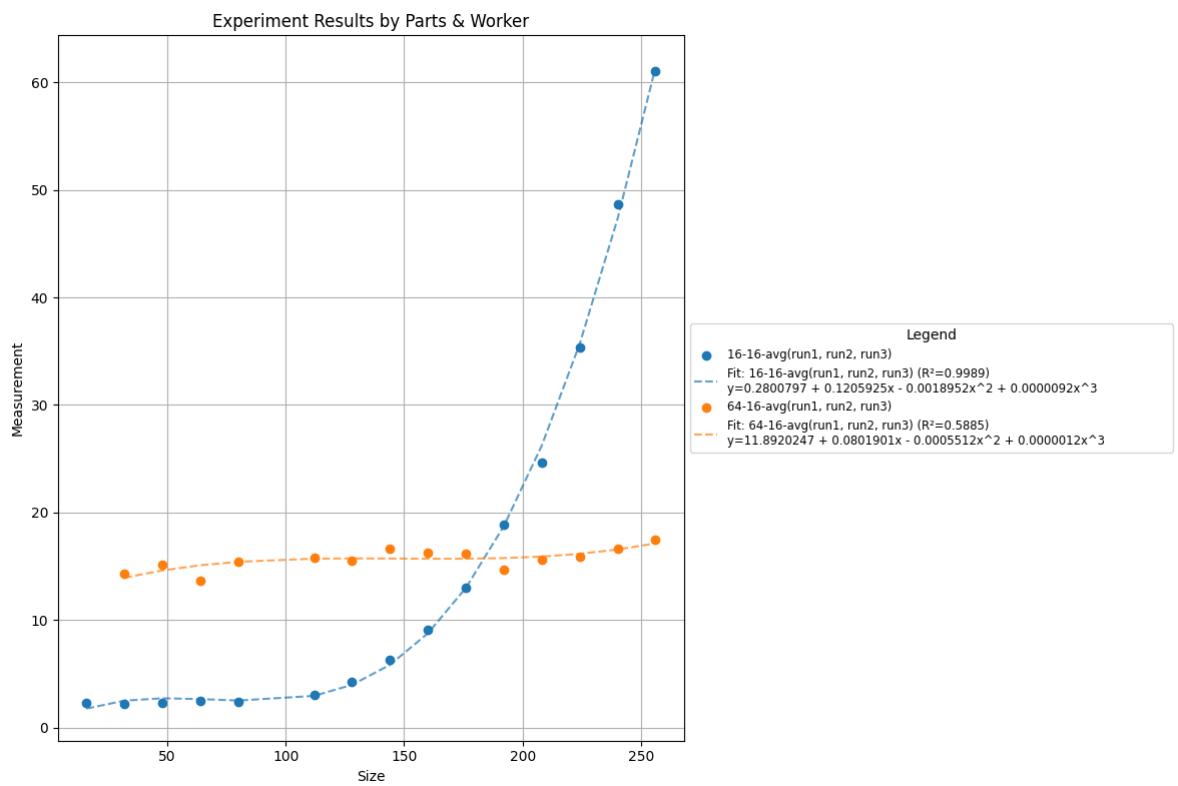


Abbildung 5.11.: 16 Worker Vergleich

5.5. Diskussion

Bei Nutzung des WFC-Algorithmus stehen dem Nutzer viele Optionen zur Verfügung. Doch ist es nicht einfach zu wissen, welche Einstellungen gewählt werden sollten, um eine optimale Performance zu erreichen. Allerdings gibt es ein paar Richtlinien, an denen sich orientiert werden kann. Zuerst sollte eine möglichst große Anzahl an Parts gewählt werden, da die benötigte Zeit, wie in Abschnitt (5.4) beschrieben, langsamer ansteigt, je größer die Verteilung ist. Dabei zu beachten ist jedoch, dass, wie in Abbildung 5.10 zu sehen, einfach die größtmögliche Verteilung zu wählen, auch nicht die beste Option ist, da die Performance für die Berechnung kleinerer Maps in diesem Fall nicht optimal ist. Dieses Ergebnis ist bemerkenswert, da zu erwarten wäre, dass es eine optimale Chunkgröße gibt, welche durch das Verhältnis von der Eingrenzung der Kaskadierung und des Overheads von Networking und Distributor die Performance optimiert. Jedoch ist dies nicht der Fall. Womöglich liegt dies daran, dass die Arbeit, die der Distributor und das Netzwerk leisten müssen, schlechter als, wie vorher angenommen, linear skaliert. Und somit lohnt es sich, die Größe der Chunks mit der Mapgröße zu erhöhen, um diesen Effekt auszugleichen. Die Bestimmung der optimalen Chunkgröße ist eine schwerere Aufgabe als angenommen. Um eine definitive Aussage hierzu treffen zu können, werden mehr Daten benötigt (siehe Abschnitt Limitationen 5.6.4).

In den meisten praktischen Anwendungsfällen steht eine limitierte Hardware-Performance zur Verfügung, womit die Anzahl der Worker im Vorhinein festgelegt ist. Nun gilt es, eine optimale Anzahl an Parts zu bestimmen, in die eine Map aufgeteilt werden soll. Diese Frage ist auch von der oben beschriebenen Wahl der Chunkgröße abhängig. Somit ergibt sich das in Abbildung 5.11 zu sehende Diagramm. Wie hier zu erkennen, ist die beste Wahl an Parts auch von der Kantenlänge abhängig, da diese direkten Einfluss auf die Chunkgröße hat und es immer einen Break-even-Point gibt, ab dem sich eine erhöhte Anzahl an Parts mehr lohnt als die optimale Nutzung von Hardware-Ressourcen, wie in Abbildung 5.6 bis 5.9 zu sehen, bei denen die 1 zu 1 Verteilung immer die beste Performance liefert.

5.6. Limitation

5.6.1. Distributor

Eine starke performance-technische Limitation in diesem Projekt stellt der Distributor dar, da dieser nicht skaliert werden kann. Während alle vorberechneten Chunks unter den skalierbaren Worker-Instanzen aufgeteilt werden, muss die Arbeit der

Aufteilung selbst, inklusive der Berechnung aller Ränder, immer allein vom Distributor durchgeführt werden. Wäre der Distributor ebenfalls skalierbar, könnte gerade bei besonders großen Maps eine deutlich bessere Performance erzielt werden.

5.6.2. Quadratische Maps

In dieser Version der Software muss eine Map zwingend quadratisch sein. Durch eine komplexere Implementierung des Distributors könnte eine flexiblere Möglichkeit zur Parallelisierung erzielt werden, mit der auch nicht-quadratische Maps zerteilt werden können.

5.6.3. Monitoring

Während der Evaluation ist aufgefallen, dass bestimmte Probleme durch mangelnde Informationen während einer Messung nicht optimal nachvollzogen werden können. Während das RabbitMQ Management (4.3.5) auch während einer Messung eine ausführliche Übersicht der Message-Queue ermöglicht, kann die interne Arbeit des Distributors sowie der einzelnen Worker nicht umfassend eingesehen werden. Handelt es sich beispielsweise um die Generierung einer besonders großen Map oder um die Berechnung eines besonders großen Chunks, kann nicht überprüft werden, ob die Berechnung lediglich noch andauert oder ob ein Fehler vorliegt. Eine zentrale, von außen abrufbare Fortschrittsanzeige der Worker sowie des Distributors würde diesem Umstand entgegenwirken.

5.6.4. Aussagekraft der Daten

Trotz des Zugriffs auf ein leistungsstarkes Rechner-Cluster, welches für dieses Projekt zur Verfügung gestellt wurde, sind die Messdaten für viele definitive Aussagen nicht ausreichend. Da der Algorithmus grundsätzlich und unabänderbar polynomial skaliert, dauern die Messungen mit steigender Kantenlänge schnell sehr lange. Dazu kamen Probleme mit der Messmethode selbst und die zwischenzeitliche Instabilität der Software (siehe Kapitel 4.8.5), bevor diese robuster gemacht wurde. Doch mit mehr Zeit und bestenfalls mehr Rechenleistung wären hier definitive Aussagen durchaus möglich gewesen.

5.6.5. Parallelisierung

Die Methode der Parallelisierung, welche für dieses Projekt erstmalig verwendet wurde, hat in Bezug auf die Performance gute Ergebnisse geliefert. Wie in den Abbildungen 5.4 und 5.9 (Abschnitt 5.4) zu erkennen ist, ist mithilfe der Parallelisierung eine deutlich bessere Performance zu erreichen als mit dem sequenziellen Algorithmus. Jedoch hat der parallele Algorithmus ein unvorhergesehenes Problem, wie in Abbildungen 5.12 und 5.13 zu erkennen ist. Aufgrund dessen, dass die Ränder vorberechnet werden, sind diese somit bei der Einstürzung komplett unabhängig von der Entropie der anderen Tiles. Anders beschrieben wird der Rand der Chunks immer mit einer Entropietoleranz von unendlich berechnet. Dementsprechend bildet der Rand andere Muster als die generelle Map. Besonders deutlich wird dies, wie in den Abbildungen zu erkennen, wenn die Entropietoleranz für die Map sehr klein ist, in diesem Fall 0. Entsprechend wird der Effekt kleiner mit höherer Entropietoleranz oder mit höherer Verteilung und entsprechend mehr Rand auf der Map.

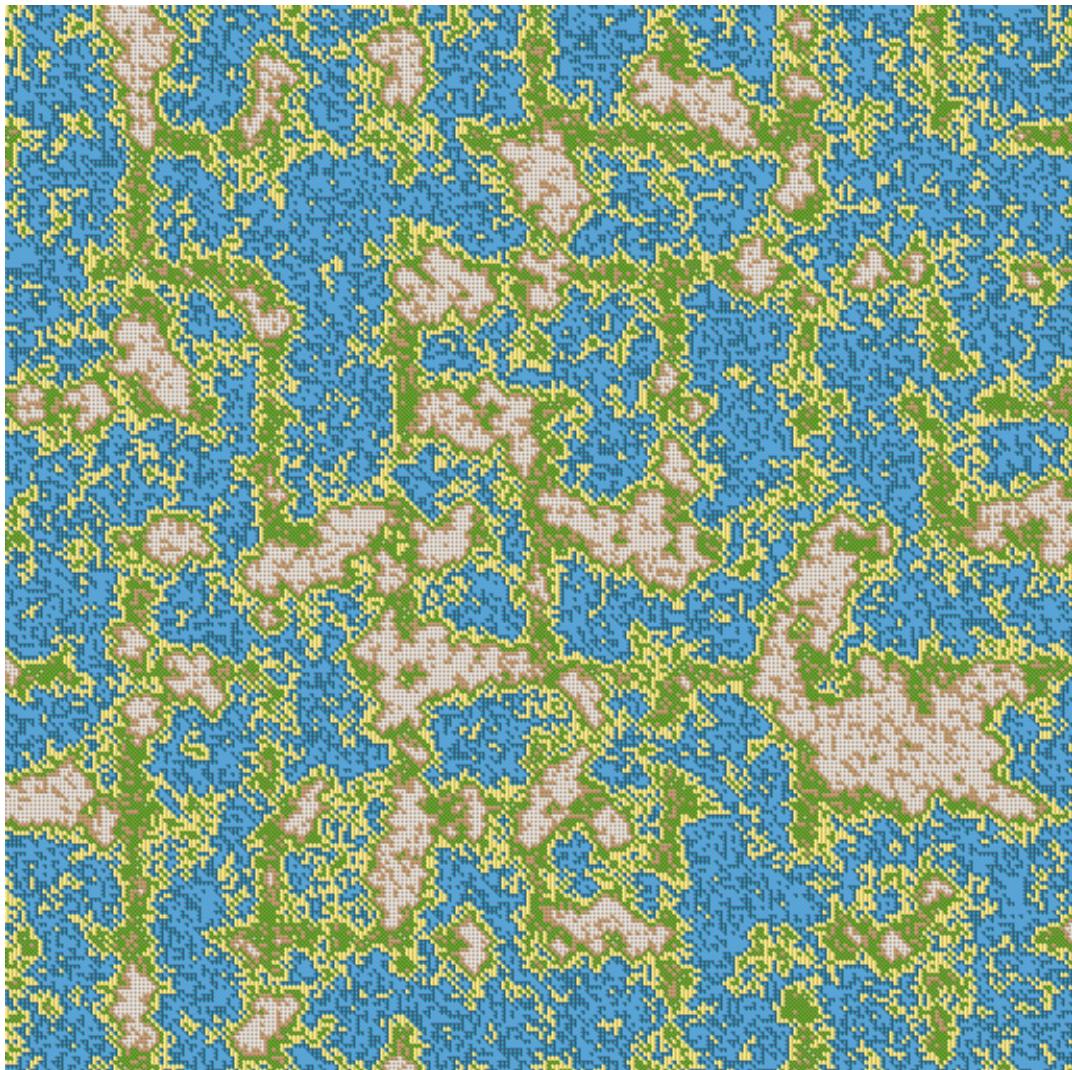


Abbildung 5.12.: Beispiel einer Map, in der die Schnittkanten der Parallelisierung mit 64 Parts erkennbar sind

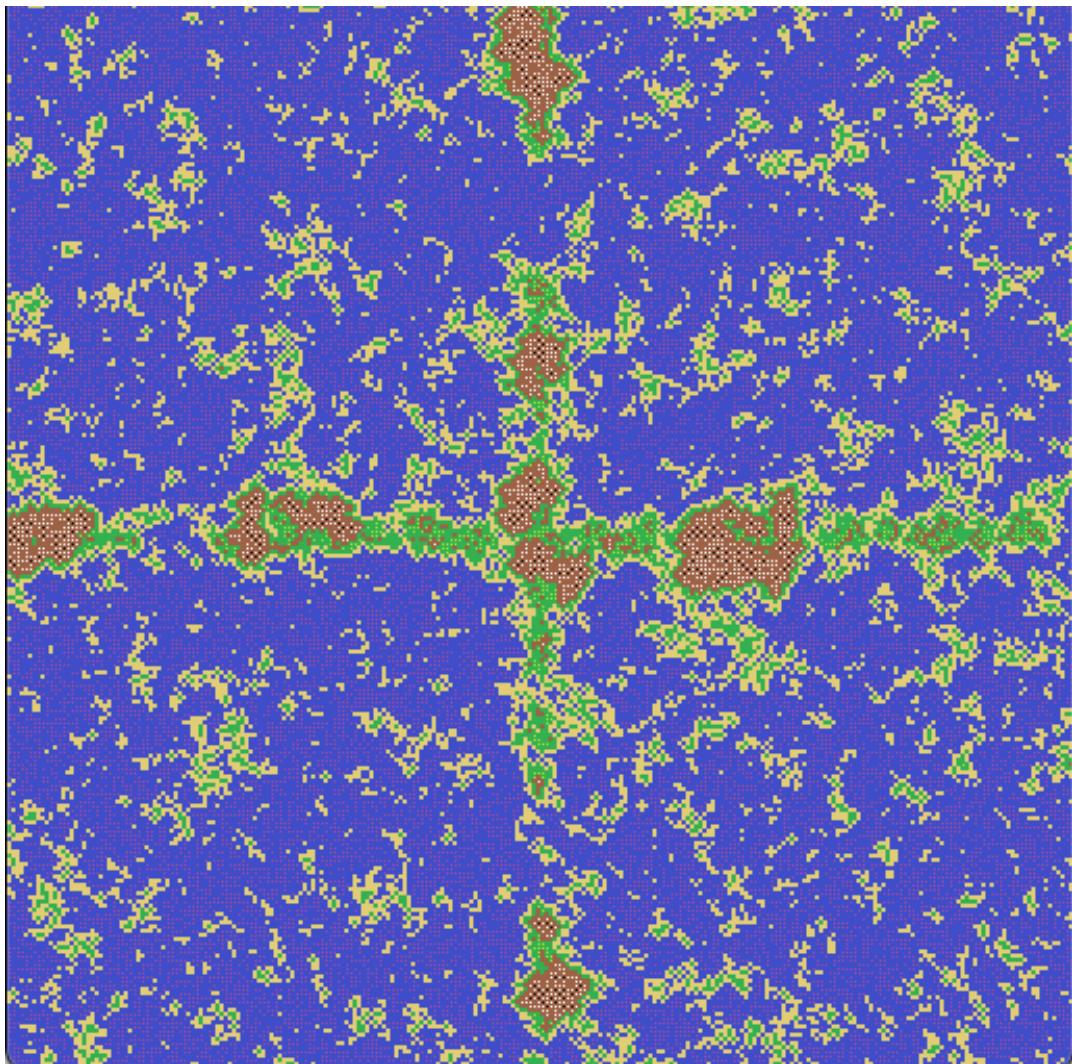


Abbildung 5.13.: Beispiel einer Map, in der die Schnittkanten der Parallelisierung mit 4 Parts erkennbar sind

5.7. Beispiele

In diesem Abschnitt werden verschiedene Beispielergebnisse unterschiedlicher Maps gezeigt (Abbildungen 5.14 - 5.21). Die Beschreibungen der Abbildungen enthalten Informationen über Größe, Verteilung und verwendeter Entropietoleranz. Weitere Beispiele können im Git-Repository eingesehen werden (Anhang A.4).

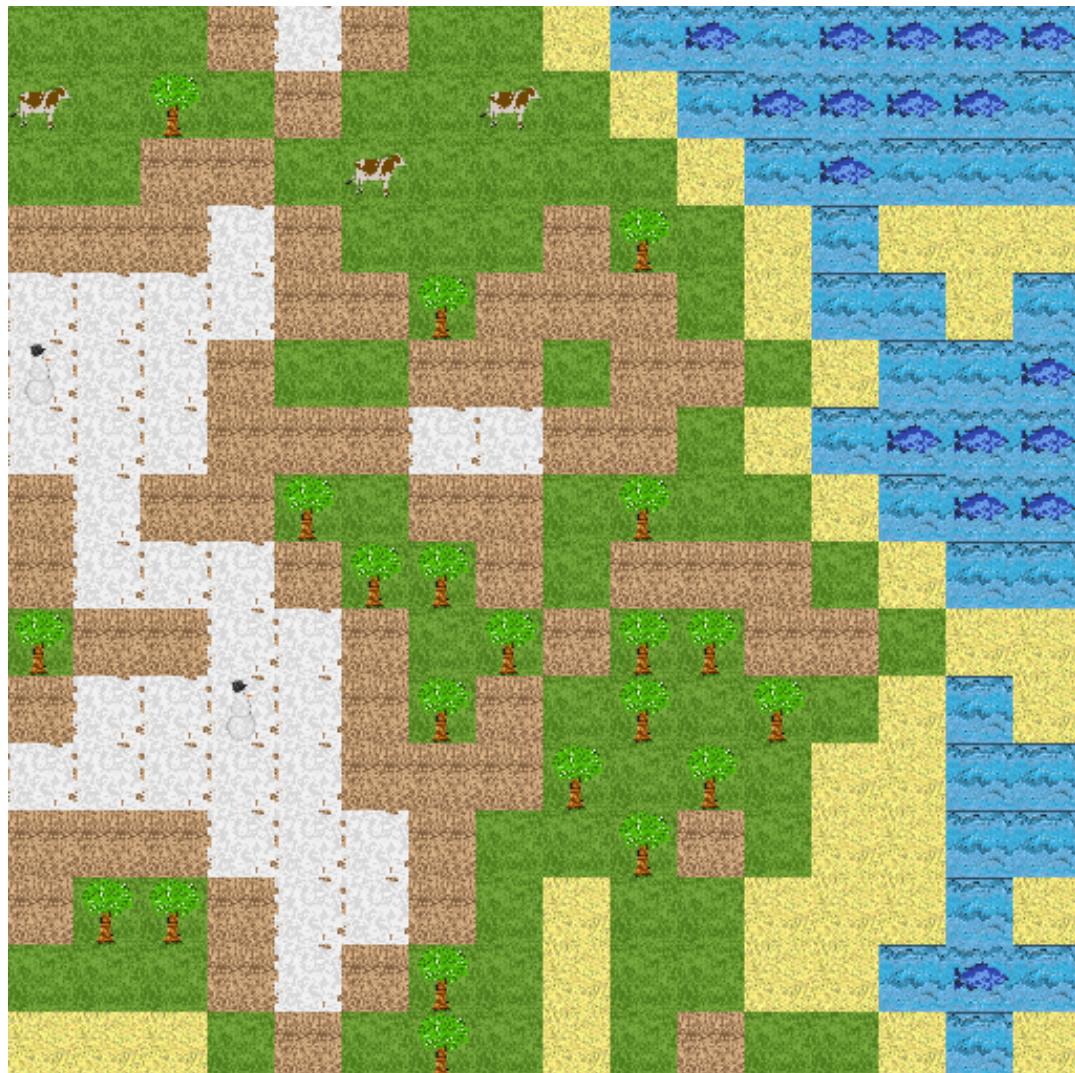


Abbildung 5.14.: Beispiel einer Map mit Kantenlänge 16, unverteilt, mit einer Entropietoleranz von 5

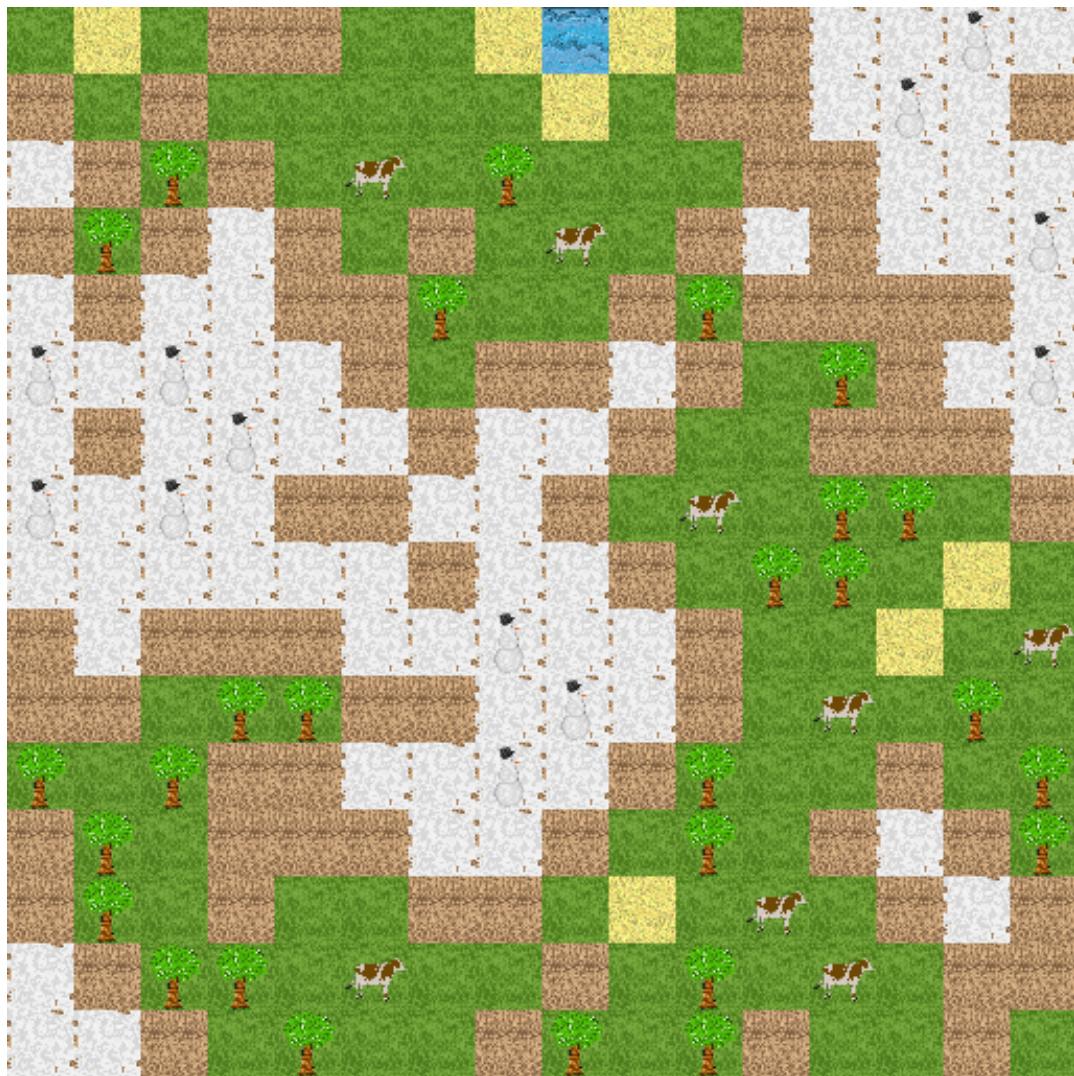


Abbildung 5.15.: Beispiel einer Map mit Kantenlänge 16, in 4 Parts, mit einer Entropietoleranz von 5



Abbildung 5.16.: Beispiel einer Map mit Kantenlänge 64, in 4 Parts, mit einer Entropietoleranz von 1

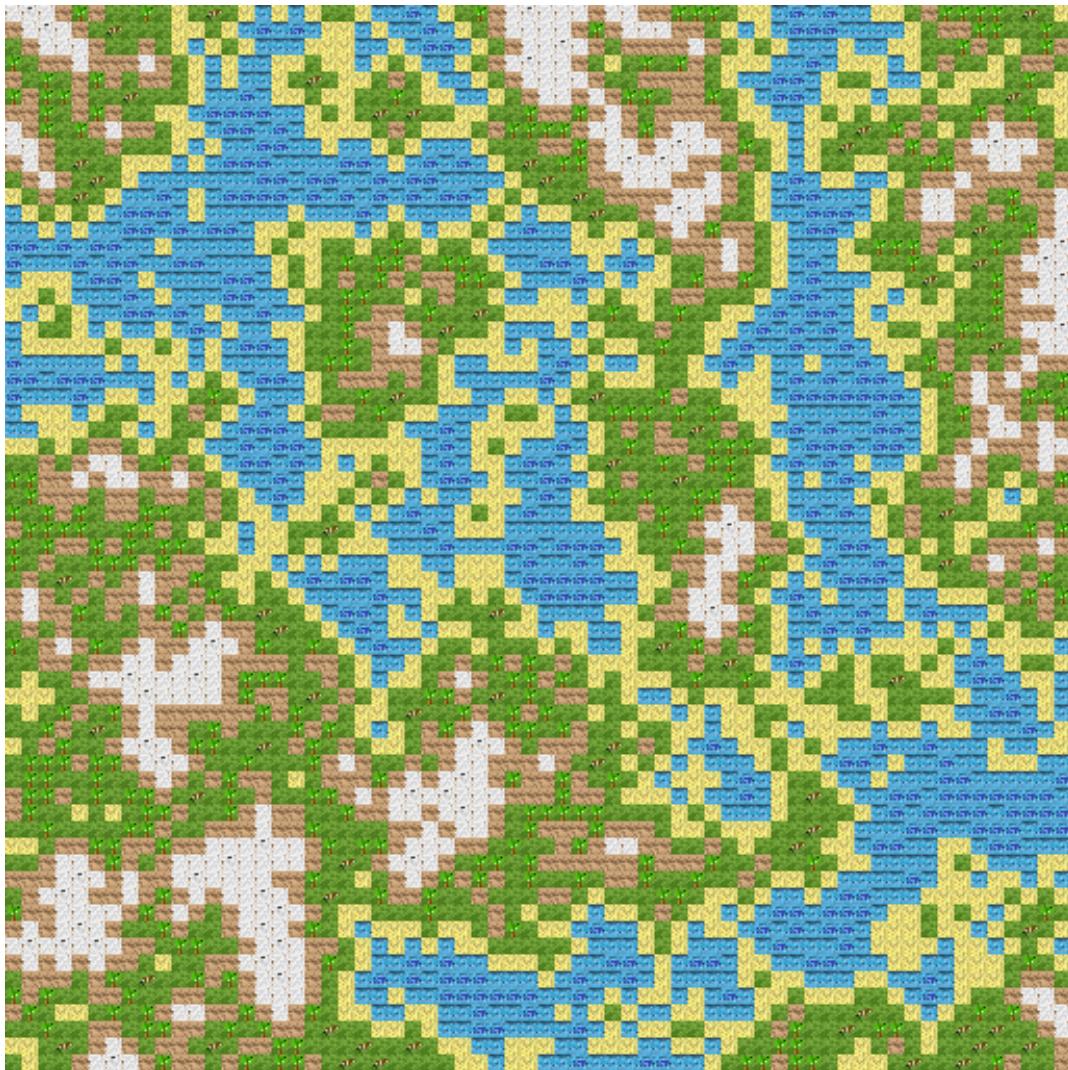


Abbildung 5.17.: Beispiel einer Map mit Kantenlänge 64, in 16 Parts, mit einer Entropietoleranz von 5

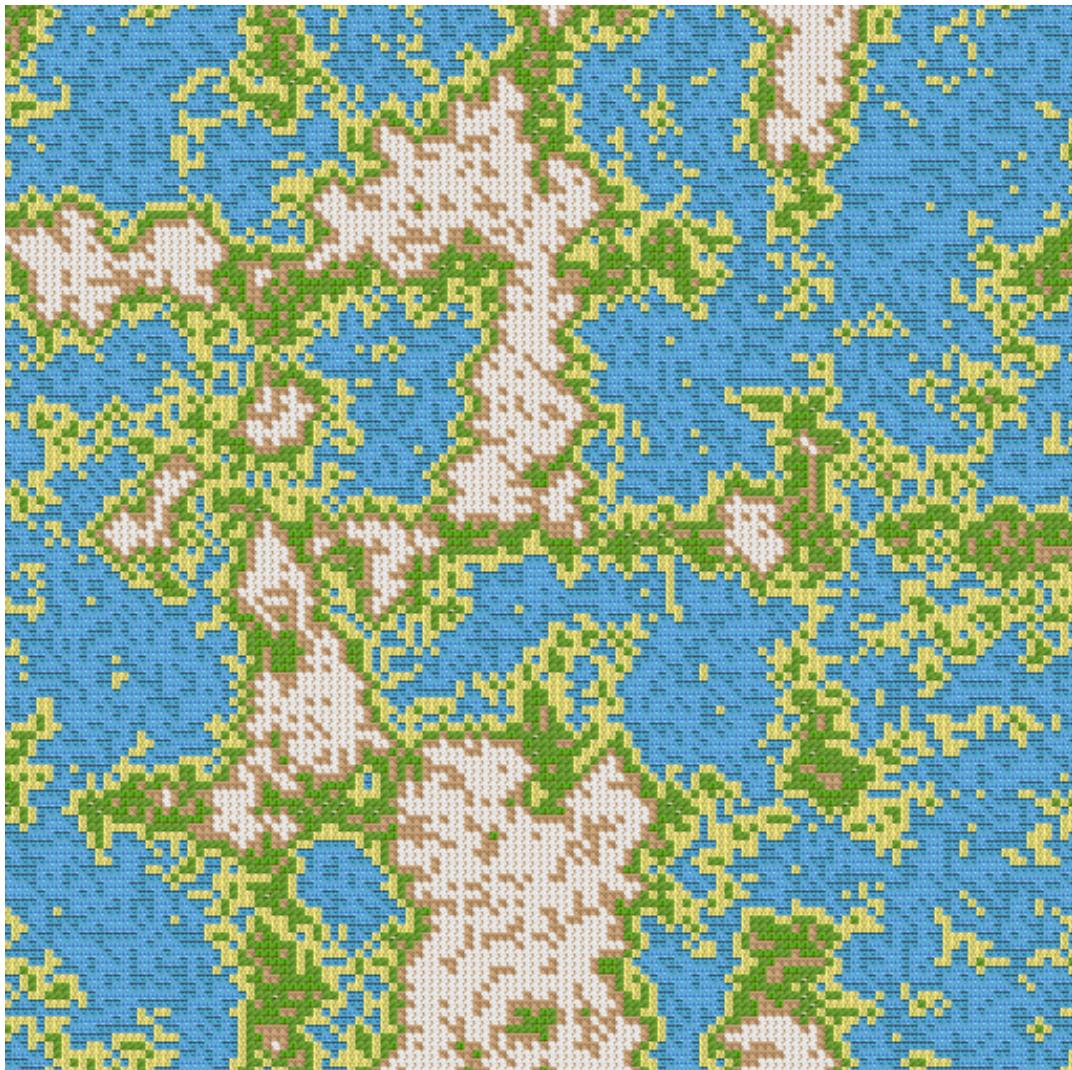


Abbildung 5.18.: Beispiel einer Map mit Kantenlänge 128, in 16 Parts, mit einer Entropietoleranz von 0

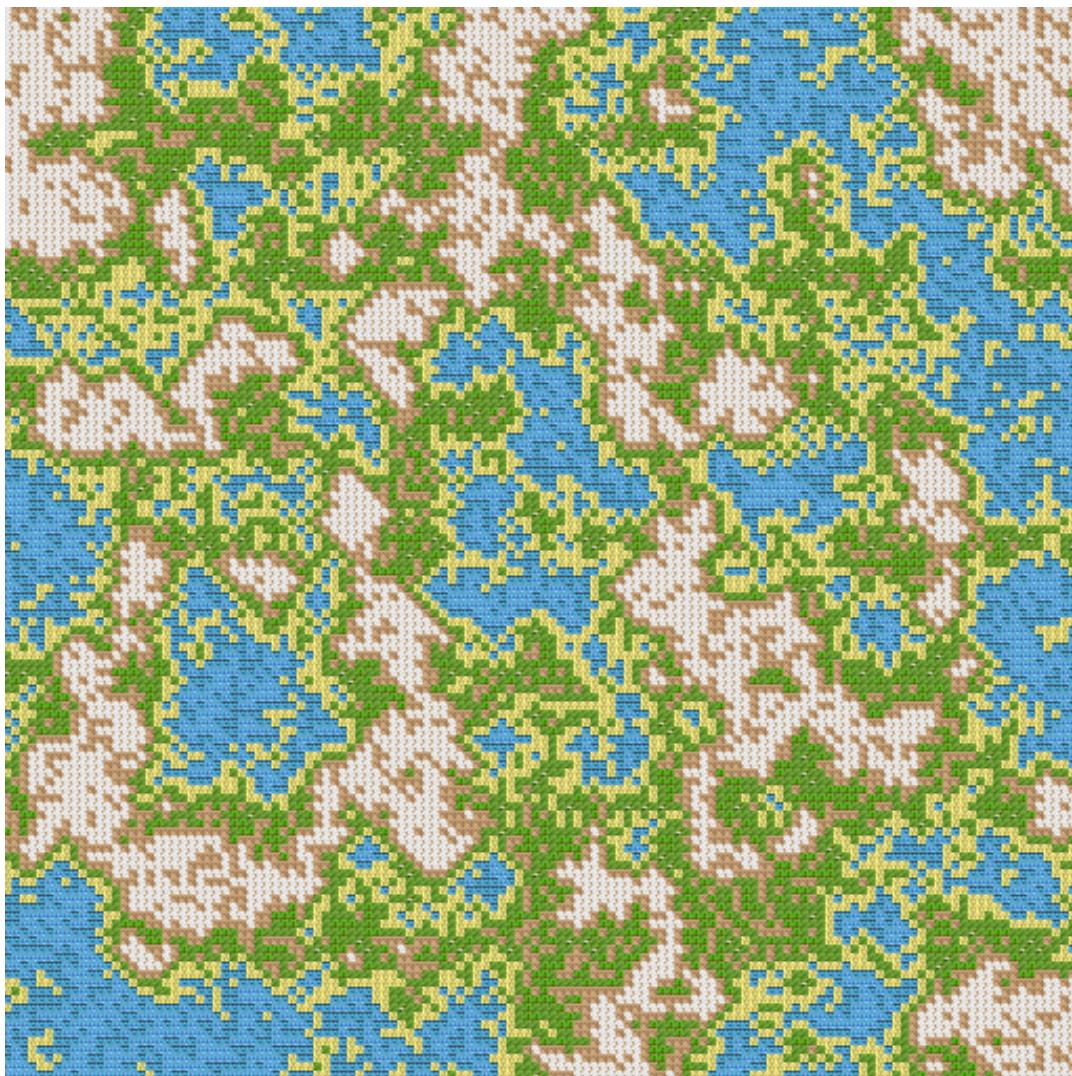


Abbildung 5.19.: Beispiel einer Map mit Kantenlänge 128, in 64 Parts, mit einer Entropietoleranz von 0

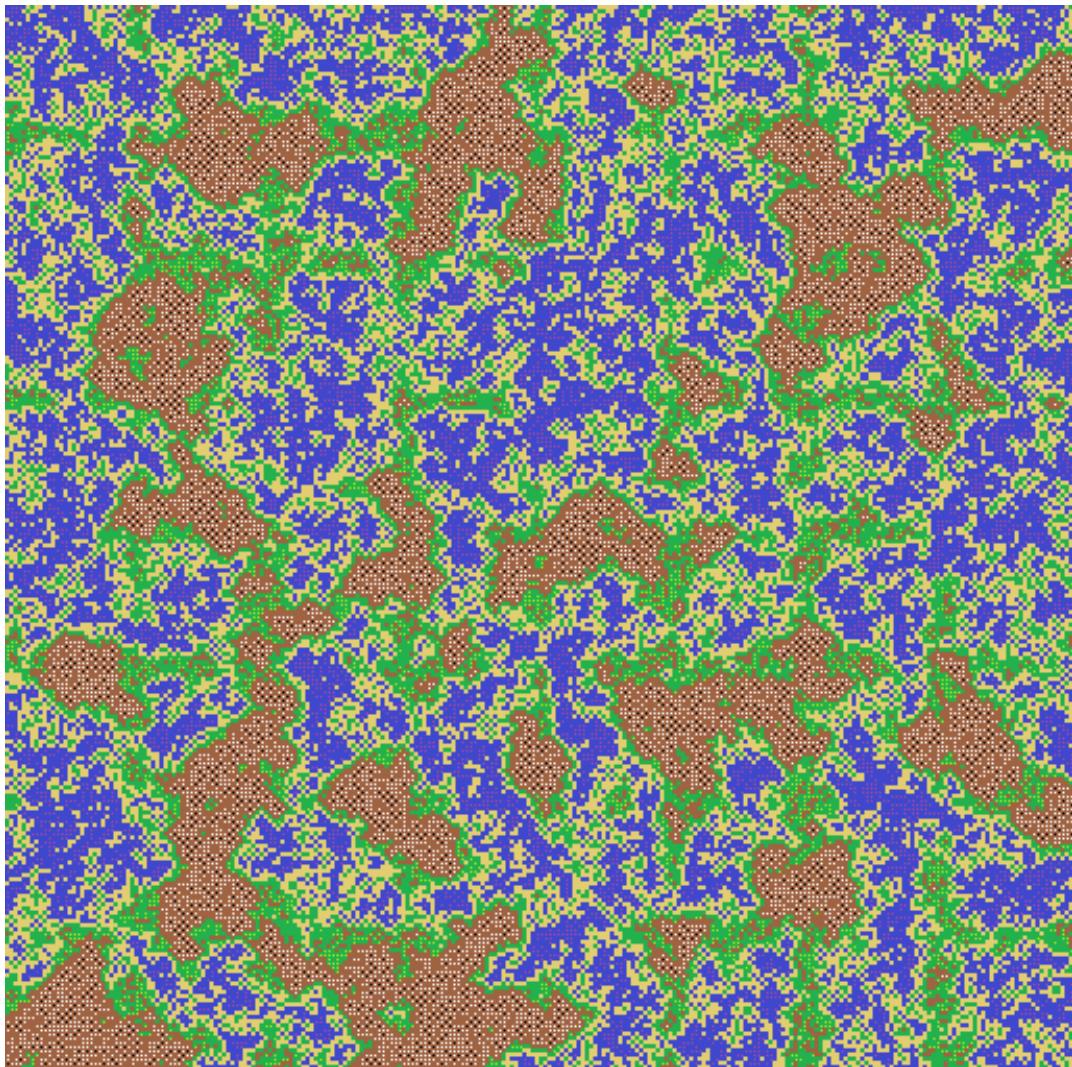


Abbildung 5.20.: Beispiel einer Map mit Kantenlänge 256, in 64 Parts, mit einer Entropietoleranz von 1

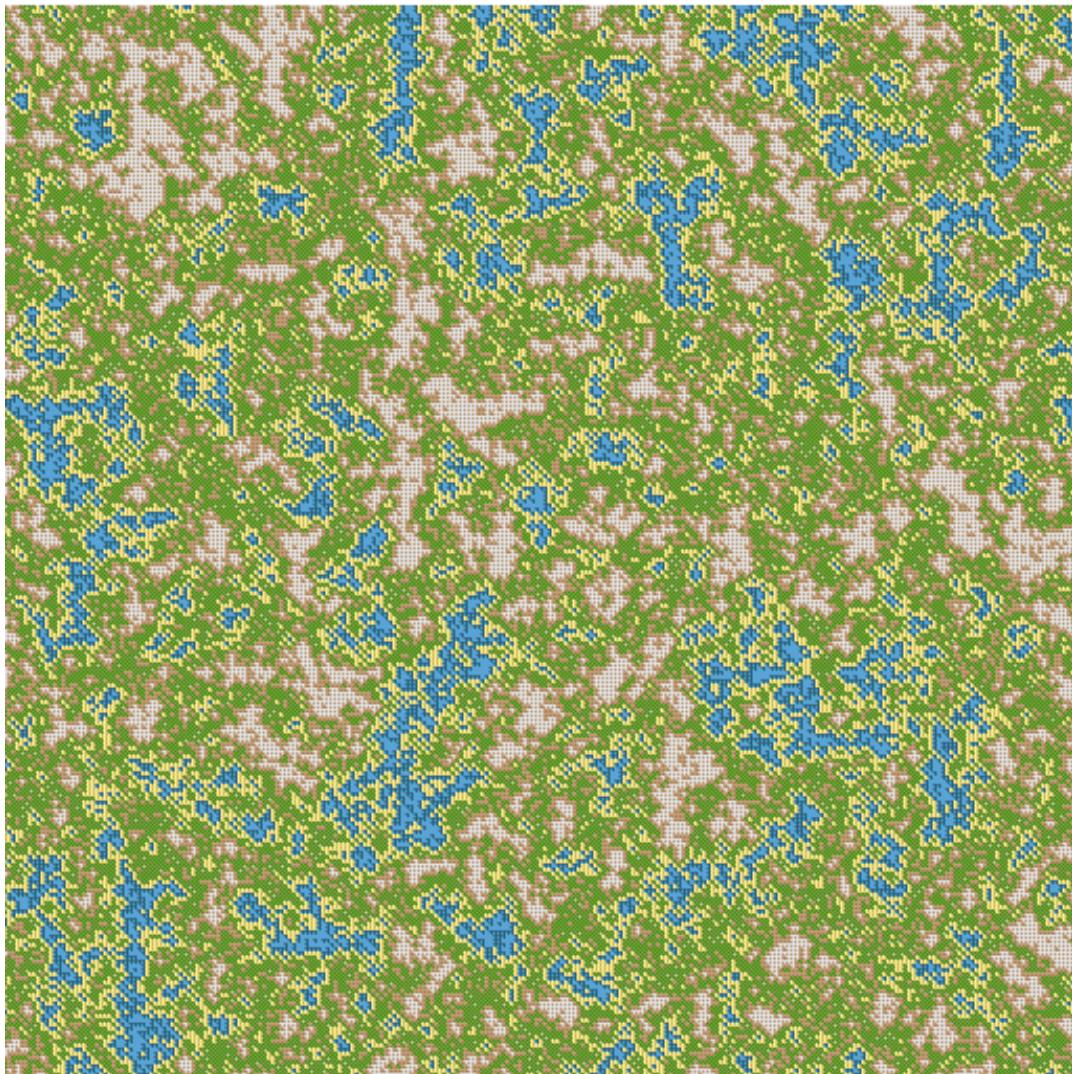


Abbildung 5.21.: Beispiel einer Map mit Kantenlänge 256, in 64 Parts, mit einer Entropietoleranz von 5

6. Fazit

6.1. Zusammenfassung

Ziel dieser Arbeit war die Entwicklung einer verteilten Softwarelösung zur prozeduralen Generierung von zweidimensionalen Landkarten unter Verwendung des Wave Function Collapse Algorithmus (WFC). Dabei wurde ein besonderes Augenmerk auf die Parallelisierbarkeit des ursprünglich sequenziell arbeitenden Algorithmus gelegt. Durch die Umsetzung einer Microservice-Architektur innerhalb eines Kubernetes-Clusters konnte ein System realisiert werden, das eine flexible und skalierbare Generierung der Landkarten ermöglicht.

Die zentrale Herausforderung bei der Parallelisierung bestand in der Zerlegung der Karten in berechenbare Abschnitte, ohne dabei die Konsistenz der Map zu gefährden. Dies wurde durch die Vorberechnung der einzelnen Ränder und die Verwendung geeigneter Regelwerke realisiert. Die Implementierung erlaubt eine parallele Bearbeitung der einzelnen Abschnitte durch mehrere Worker-Instanzen, wobei die Kommunikation über RabbitMQ koordiniert wurde.

Im Rahmen der Evaluation zeigte sich, dass die gewählte Strategie zur Parallelisierung funktioniert. Die Zeiten zur Generierung konnten mit steigender Worker-Anzahl vor allem bei besonders großen Maps signifikant reduziert werden. Es wurde jedoch auch deutlich, dass eine sinnvolle Anzahl an Abschnitten und eine passende Entropietoleranz entscheidend für die Qualität und Erfolgsrate der Generierungen sind. Hier zeigte sich ein klassisches Spannungsfeld zwischen Performance, Konsistenz und Komplexität.

Die systematische Erfassung der Zeitdaten sowie deren spätere Analyse trugen zur fundierten Bewertung der Performance bei.

Insgesamt konnte gezeigt werden, dass der WFC-Algorithmus unter bestimmten Bedingungen parallelisierbar ist.

6.2. Ausblick

Die Ergebnisse dieser Arbeit ermöglichen verschiedene Ansätze zur Weiterentwicklung:

Nicht-quadratische Maps

Eine der naheliegendsten und sinnvollsten Erweiterungen wäre es, den Distributor so anzupassen, dass dieser auch Maps verteilen kann, welche nicht quadratisch sind. Die Grundlage hierfür ist bereits im Algorithmus sowie in den Regeln und Restriktionen eingebaut. Dies würde die Anzahl an Optionen für die Mapgröße, Chunkgröße und Parts stark erhöhen. Auch für genauere Messungen wäre diese Erweiterung sehr nützlich.

Adaptive Chunkgröße

Eine dynamische Anpassung der Chunkgröße zur Laufzeit könnte für eine noch effizientere Aufwandsverteilung sorgen, insbesondere bei nicht-quadratischen Maps.

Erweiterung des Tilesets

Auf Grundlage dieser Arbeit können einfach Änderungen und Erweiterungen an den Tilesets, Regeln und Restriktionen, welche in diesem Projekt eingeführt wurden, gemacht werden.

Nutzerinteraktion

Der in diesem Projekt entwickelte Algorithmus muss aufgrund der Parallelisierung stabil gegenüber Abweichungen der Einsturzordnung sein. Dies könnte dem Nutzer ermöglichen, selber in den Algorithmus einzutreten, um die Ergebnisse noch besser kontrollieren und die Maps somit besser gestalten zu können.

Verschachtelung des Distributors

Anstatt, wie in diesem Projekt, nur einen Distributor-Service zu verwenden, der zu Beginn der Generierung angesprochen wird, könnte es sich lohnen, die Arbeit des Distributors zu verteilen. Ein interessanter Ansatz dafür wäre es, den Distributor zu verschachteln, indem die Abschnitte, in die der Distributor die Map zerteilt, erneut an einen Distributor zur weiteren Aufteilung weitergegeben werden, bis die Abschnitte eine bestimmte Mindestgröße erreicht haben. So kann auch der Distributor parallelisiert werden und somit eines der grundlegenden Probleme, die in dieser Arbeit aufgetreten sind, adressiert werden.

Literatur

- AWS, Amazon (2025). *Was ist ein Kubernetes-Cluster*. URL: <https://aws.amazon.com/de/what-is/kubernetes-cluster/> (besucht am 01.06.2025).
- Baier, Jonathan (2017). *Getting Started with Kubernetes*. 2. Auflage. Packt Publishing.
- Erickson, Jeffrey (2025). *MySQL: Was es ist und wie es verwendet wird*. URL: <https://www.oracle.com/de/mysql/what-is-mysql/> (besucht am 01.06.2025).
- Grama, Ananth (2003). *Introduction to Parallel Computing*. Addison Wesley.
- Gumin, Maxim (2025). *WaveFunctionCollapse*. URL: <https://github.com/mxgnn/WaveFunctionCollapse> (besucht am 01.06.2025).
- Gupta, Pradeep (2025). *CUDA Refresher: The CUDA Programming Model*. URL: <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/> (besucht am 01.06.2025).
- Hausenblas, Michael (2019). *Programming Kubernetes*. 1. Auflage. O'Reilly.
- Heaton, Robert (2025). *The Wavefunction Collapse Algorithm explained very clearly*. URL: <https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/> (besucht am 01.06.2025).
- Hightower, Kelsey (2022). *Kubernetes: Up & Running*. 3. Auflage. O'Reilly.
- IBM (2025a). *Was ist eine Nachrichtenwarteschlange*. URL: <https://www.ibm.com/de-de/think/topics/message-queues> (besucht am 01.06.2025).
- (2025b). *Was ist Virtualisierung*. URL: <https://www.ibm.com/de-de/topics/virtualization> (besucht am 01.06.2025).
- Kubernetes (2025a). *Cluster Architecture*. URL: <https://kubernetes.io/docs/concepts/architecture/> (besucht am 01.06.2025).
- (2025b). *Deployments*. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/> (besucht am 01.06.2025).
- (2025c). *Installieren und konfigurieren von kubectl*. URL: <https://kubernetes.io/de/docs/tasks/tools/install-kubectl/> (besucht am 01.06.2025).
- (2025d). *Kubeadm*. URL: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/> (besucht am 01.06.2025).
- (2025e). *Kubernetes Komponenten*. URL: <https://kubernetes.io/de/docs/concepts/overview/components/> (besucht am 01.06.2025).
- (2025f). *Pods*. URL: <https://kubernetes.io/de/docs/concepts/workloads/pods/> (besucht am 01.06.2025).
- (2025g). *Was ist Kubernetes*. URL: <https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/> (besucht am 01.06.2025).

- Luca Stamos Stefan Steinhauer, Marc Kevin Zenzen (2022). „VDM Dokumentation“. In: *TH Köln Campus Gummersbach*.
- Lukša, Marko (2022). *Kubernetes in Action*. 2. Auflage. Manning.
- Mattson, Timothy G (2008). *Patterns for Parallel Programming (Software Patterns Series)*. Addison Wesley.
- Merrell, Paul (2021). „Model Synthesis: A General Procedural Modeling Algorithm“. In: *IEEE Transactions on Visualization and Computer Graphics*. Zugriff am 1. Juni 2025. URL: <https://paulmerrell.org/wp-content/uploads/2021/06/tvcg.pdf>.
- (2022). „Example-Based Model Synthesis“. In: *University of North Carolina at Chapel Hill*. Zugriff am 1. Juni 2025. URL: https://paulmerrell.org/wp-content/uploads/2022/03/model_synthesis.pdf.
- Parker, Joseph (2025). *Generating Worlds With Wave Function Collapse*. URL: <https://www.procjam.com/tutorials/wfc/> (besucht am 01.06.2025).
- RedHat (2025). *Was ist ein Kubernetes-Cluster*. URL: <https://www.redhat.com/de/topics/containers/what-is-a-kubernetes-cluster> (besucht am 01.06.2025).
- Robotbugs (2025). *Fast way of counting non-zero bits in positive integer*. URL: <https://stackoverflow.com/questions/9829578/fast-way-of-counting-non-zero-bits-in-positive-integer> (besucht am 01.06.2025).

A. Git-Repository

A.1. Git-Repository

<https://github.com/pfropfen/wfc>

A.2. VDM Dokumentation

<https://github.com/pfropfen/wfc/blob/main/LIB/VDM%20Dokumentation.pdf>

A.3. Messergebnisse

<https://github.com/pfropfen/wfc/tree/main/MESSREIHEN>

A.4. Beispielbilder

<https://github.com/pfropfen/wfc/tree/main/PICS/MAPS>

B. Messreihen

B.1. Messreihen

<https://github.com/pfropfen/wfc/blob/main/messreihen.csv>

Die Messtabelle (Abbildung B.1 - B.11) setzt sich aus folgenden Parametern zusammen:

- **Row**: Nummer der Messung
- **SIZE**: Kantenlänge der Map in Tiles
- **PARTS**: Anzahl der Abschnitte der Map
- **WORKER**: Anzahl der beteiligten Worker
- **Run1**: Erste gemessene Zeit in Millisekunden
- **Run2**: Zweite gemessene Zeit in Millisekunden
- **Run3**: Dritte gemessene Zeit in Millisekunden

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
1	16	1	1	315	311	334
2	16	4	1	2001	1880	1737
3	16	4	2	972	998	867
4	16	4	4	519	401	494
5	16	16	4	3354	3270	3119
6	16	16	8	2891	2639	2952
7	16	16	16	2261	2220	2305
8	32	1	1	4951	2782	3869
9	32	4	1	2458	2306	2234
10	32	4	2	1322	1261	1165
11	32	4	4	491	484	675
12	32	16	4	3453	3428	3622
13	32	16	8	3229	2694	2841
14	32	16	16	2271	2020	2320
15	32	64	16	15252	12730	14913
16	32	64	32	12078	12446	11767
17	32	64	64	8464	8368	8772
18	48	1	1	20736	17865	19557
19	48	4	1	4368	3843	3806
20	48	4	2	1918	2116	2307

Abbildung B.1.: Ausschnitt 1 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
21	48	4	4	1112	1202	1140
22	48	16	4	3312	3037	3504
23	48	16	8	3280	3266	3126
24	48	16	16	2355	2210	2395
25	48	36	9	7339	7455	6464
26	48	36	18	8029	7916	8275
27	48	36	36	4433	4744	4692
28	48	64	16	13886	15968	15477
29	48	64	32	12319	12389	12041
30	48	64	64	8501	8766	8960
31	64	1	1	78976	56399	77452
32	64	4	1	12993	13675	14164
33	64	4	2	7277	6199	6259
34	64	4	4	3980	3511	3488
35	64	16	4	3631	3648	3779
36	64	16	8	3371	2818	2965
37	64	16	16	2615	2470	2447
38	64	64	16	13291	16387	11256
39	64	64	32	12506	12181	13009
40	64	64	64	8749	9224	9165

Abbildung B.2.: Ausschnitt 2 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
41	80	1	1	162974	159558	152876
42	80	4	1	32981	30913	31873
43	80	4	2	18261	17374	17354
44	80	4	4	8845	8780	8859
45	80	16	4	3970	3723	4142
46	80	16	8	3418	3371	3056
47	80	16	16	2467	2309	2423
48	80	64	16	15147	15091	16024
49	80	64	32	12974	12135	12807
50	80	64	64	8783	8327	8689
51	96	1	1	309216	292448	382545
52	96	4	1	68132	76430	74483
53	96	4	2	39737	36412	37216
54	96	4	4	19750	17483	17763
55	96	36	9	7815	6384	8000
56	96	36	18	7378	8004	8889
57	96	36	36	4726	4893	4906
58	112	1	1	751077	577960	807006
59	112	4	1	123691	140341	146334
60	112	4	2	70340	70213	69946

Abbildung B.3.: Ausschnitt 3 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
61	112	4	4	37688	36220	32589
62	112	16	4	7917	8214	7436
63	112	16	8	5457	5129	4999
64	112	16	16	2961	3088	3025
65	112	64	16	17093	14926	15378
66	112	64	32	12561	12765	13092
67	112	64	64	8571	9133	8718
68	112	196	49	44911	44733	44451
69	112	196	98	40982	40951	41631
70	112	196	196	35770	34587	35128
71	112	256	64	48543	60636	48330
72	112	256	128	55954	56492	54640
73	112	256	256	47959	48092	47688
74	128	1	1	1053006	876351	1450681
75	128	4	1	207096	227946	250196
76	128	4	2	126223	116076	119414
77	128	4	4	63911	62161	60976
78	128	16	4	11759	12937	12319
79	128	16	8	7792	7446	7021
80	128	16	16	4705	4574	3474

Abbildung B.4.: Ausschnitt 4 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
81	128	64	16	16364	16725	13570
82	128	64	32	12884	13076	12726
83	128	64	64	8798	8777	9152
84	128	256	64	51058	49916	56190
85	128	256	128	54470	53035	56020
86	128	256	256	47033	47834	48024
87	144	1	1	1731592	1360242	1438063
88	144	4	1	376671	385812	391282
89	144	4	2	194888	191178	185335
90	144	4	4	104423	101729	94140
91	144	16	4	21623	19917	19375
92	144	16	8	12061	10870	10379
93	144	16	16	6018	6123	6675
94	144	64	16	16805	16816	16327
95	144	64	32	12443	12542	12792
96	144	64	64	9125	8966	9091
97	144	144	36	30762	26566	27717
98	144	144	72	30423	29511	29247
99	144	144	144	24456	24671	24866
100	144	256	64	59836	61703	56669

Abbildung B.5.: Ausschnitt 5 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
101	144	256	128	55278	55360	55231
102	144	256	256	48841	48221	47979
103	160	1	1	2208680	2682021	2435507
104	160	4	1	562834	600945	650544
105	160	4	2	280700	321010	299481
106	160	4	4	166319	154323	155278
107	160	16	4	33317	30914	32607
108	160	16	8	17262	17511	15779
109	160	16	16	9305	8965	9082
110	160	64	16	16381	15467	16857
111	160	64	32	12917	13025	12753
112	160	64	64	9236	9404	9140
113	160	100	25	18430	22630	19495
114	160	100	50	20250	20148	20421
115	160	100	100	15349	14974	15320
116	160	256	64	49278	59765	53099
117	160	256	128	54963	56008	56380
118	160	256	256	49347	47470	48075
119	176	1	1	4972273	5376737	3295807
120	176	4	1	860525	907887	981107

Abbildung B.6.: Ausschnitt 6 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
121	176	4	2	401317	458664	464838
122	176	4	4	238027	224979	226773
123	176	16	4	46788	45101	48341
124	176	16	8	25915	25411	24217
125	176	16	16	12627	12747	13524
126	176	64	16	16396	16638	15482
127	176	64	32	12400	13455	12603
128	176	64	64	9410	9417	9554
129	176	256	64	56150	56650	56060
130	176	256	128	55049	56964	59101
131	176	256	256	47169	48324	48543
132	192	1	1	6581536	4518033	5518994
133	192	4	1	1287384	1298319	1275580
134	192	4	2	621631	642258	662204
135	192	4	4	337115	324800	306746
136	192	16	4	66661	64405	66537
137	192	16	8	38278	35133	37592
138	192	16	16	19722	17756	19081
139	192	64	16	14413	15997	13564
140	192	64	32	13368	13210	12928

Abbildung B.7.: Ausschnitt 7 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
141	192	64	64	9730	9733	9732
142	192	144	36	30469	29961	32937
143	192	144	72	28867	30518	29076
144	192	144	144	25016	24625	24891
145	192	256	64	55980	43978	55427
146	192	256	128	56243	54812	55036
147	192	256	256	47609	48836	48791
148	208	1	1	7006513	6609564	8036441
149	208	4	1	1754656	1730332	1702763
150	208	4	2	889028	938481	950842
151	208	4	4	495692	447615	434690
152	208	16	4	99839	98335	88300
153	208	16	8	52780	49603	55608
154	208	16	16	23579	25167	25251
155	208	64	16	15665	16662	14404
156	208	64	32	12891	13693	13245
157	208	64	64	9882	9841	9616
158	208	256	64	53926	49735	56864
159	208	256	128	56700	56271	55410
160	208	256	256	48539	48964	47763

Abbildung B.8.: Ausschnitt 8 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
161	224	1	1	10213501	10866091	10659430
162	224	4	1	2408281	2421887	2349265
163	224	4	2	1153527	1252561	1301903
164	224	4	4	675703	635501	618054
165	224	16	4	140953	125453	128628
166	224	16	8	71504	66209	69002
167	224	16	16	37114	35294	33717
168	224	64	16	17095	14516	15943
169	224	64	32	13403	13700	13265
170	224	64	64	9756	9773	10103
171	224	196	49	46056	45377	44469
172	224	196	98	40681	41519	40603
173	224	196	196	34268	34093	34714
174	224	256	64	55887	57752	59286
175	224	256	128	56314	58357	56836
176	224	256	256	48911	47935	48215
177	240	1	1	10863717	17188778	10766386
178	240	4	1	3099323	3325083	3156010
179	240	4	2	1627933	1592793	1717747
180	240	4	4	879376	854695	818815

Abbildung B.9.: Ausschnitt 9 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
181	240	16	4	180317	179810	179492
182	240	16	8	97072	90455	97075
183	240	16	16	51368	49192	45405
184	240	64	16	14469	19033	16519
185	240	64	32	14033	13715	14578
186	240	64	64	10471	10535	10547
187	240	100	25	23462	24549	20436
188	240	100	50	19715	20051	20286
189	240	100	100	15779	15718	15793
190	240	256	64	51562	56095	50230
191	240	256	128	57117	53721	58435
192	240	256	256	48884	48142	47599
193	256	1	1	21362694	22492321	17435302
194	256	4	1	4012519	4287079	4239871
195	256	4	2	2039797	2123840	2247751
196	256	4	4	1099653	1072927	1064565
197	256	16	4	238245	234253	244259
198	256	16	8	128264	124429	121944
199	256	16	16	64408	58757	59986
200	256	64	16	18041	17464	16868

Abbildung B.10.: Ausschnitt 10 der Messreihentabelle

Row	SIZE	PARTS	WORKER	Run1	Run2	Run3
201	256	64	32	15058	15609	15856
202	256	64	64	10680	10553	10652
203	256	256	64	51584	55462	55294
204	256	256	128	56363	57736	59080
205	256	256	256	49638	47459	47535

Abbildung B.11.: Ausschnitt 11 der Messreihentabelle