Democratizing Scalable Cloud Applications

Transactional Stateful Functions on Streaming Dataflows

arXiv:2512.17429v1 [cs.DB] 19 Dec 2025

Democratizing Scalable Cloud Applications

Transactional Stateful Functions on Streaming Dataflows

博士论文

for the purpose of obtaining the degree of doctor
at Delft University of Technology,
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates,
to be defended publicly on
Wednesday 14 January 2026 at 17.30 o'clock

by

Kyriakos PSARAKIS

Master of Science in Computer Science,
Delft University of Technology, Kingdom of the Netherlands,
and Master of Engineering in Electrical and Computer Engineering,
Technical University of Crete, Hellenic Republic,
born in Chania, Hellenic Republic.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	主席
Prof. dr. ir. G.J.P.M. Houben,	Delft University of Technology, *promotor*
Dr. A. Katsifodimos,	Delft University of Technology, *copromotor*
Independent members:	
Prof. dr. S. Madden,	Massachusetts Institute of Technology
Prof. dr. G. Smaragdakis,	Delft University of Technology
Dr. P. Carbone,	KTH Royal Institute of Technology
Dr. A. Klimović,	苏黎世联邦理工学院
Prof. dr. A. van Deursen,	Delft University of Technology, *reserve member*

SIKS Dissertation Series No. 2026-03
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.





关键词: stream processing, transactions, stateful functions, state migration, deterministic transactions, dataflow programming, microservice architectures, event-driven programming, fault tolerance

封面: Cover designed by Nicolás Oliveros

Copyright © 2026 by Kyriakos Psarakis

ISBN/EAN: 978-94-6518-185-1

一位被选举的人ronic version of this dissertation is available at: http://repository.tudelft.nl

Life is Short, Art long, Opportunity fleeting, Experience deceitful, and Judgment difficult.

Ὁ βίος βραχύς, ἡ δὲ τέχνη μακρή, ὁ δὲ καιρὸς ὀξύς, ἡ δὲ πεῖρα σφαλερή, ἡ δὲ κρίσις χαλεπή.

The Aphorisms of Hippocrates, 5th Century B.C.

目录

Summa ry **xi**

Samenv 装饰xii **i**

Περίληψη **xv**

1引言 1

1.1 从裸机到云端.3 1.2 可扩展云应用方面 .5 1.2.1 云运行时与编程模型.5 1.2.2 事务保证 .6 1.2.3 高可用性与容错 .8 1.2.4 适应性.8 1.3 什么是无服务器？. 9 1.3.1 开发者视角 .9 1.3.2 系统考量 .10 1.3.3 经济学.10 1.3.4 本论文阐释 .11 1.4 主要研究问题.11 1.5 贡献 .12 1.6 论文起源 .13 1.7 可视化大纲 .15

2 Tran事务型云应用随数据（流）而行 17

2.1 从单体应用到微服务.19 2.1.1 消息传递、幂等性与一致性 .20 2.1.2 事务与编排 .20 2.1.3 应用程序（重新）扩缩 .21 2.2 流式数据流力挽狂澜.21 2.2.1 作为架构抽象的数据流.22 2.2.2 面向事务型应用的数据流.22 2.3 Stateflo w/Styx 方法 .24 2.3.1 编程模型 .24 2.3.2 Styx 运行时 .24 2.4 相关工作.25

3 T-Statefun: Adapting Existing Approaches for Transactional Stateful Functions **27**

3.1 流式数据流上的事务 .30 3.2 预备知识.30

3.3 协调器函数与 T-Statefun API .33 3.4 事务工作流.36 3.4.1 协调器函数 .36 3.4.2 Saga 协调 .36 3.4.3 两阶段提交协调 .38 3.5 T-Statefun 的事务保证 .39 3.6 实验评估 .40 3.6.1 协调开销 .43 3.6.2 Saga 与两阶段提交的比较.44 3.6.3 可扩展性比较.47 3.6.4 微基准测试.48 3.6.5 与最先进水平的比较.48 3.7 相关工作.51 3.8 讨论与开放问题.52 3.9 结论.53

4 Stateflow：一种面向通用云应用程序的领域特定语言 55

4.1 从命令式代码到数据流 .57 4.1.1 方法概述 .58 4.1.2 编程模型与局限性 .59 4.1.3 从实体到数据流算子 .59 4.1.4 从命令式函数到数据流 .60 4.1.5 中间表示 .61 4.2 支持的数据流系统运行时 .61 4.3 初步实验.62 4.4 开放问题与机遇 .64 4.5 相关工作.65 4.6 结论.66

5 Styx: a Transactional Dataflow-Based Runtime for Stateful Functions as a Service 67

5.1 动机.69 5.1.1 有状态函数的数据流 .70 5.1.2 确定性与事务.70 5.1.3 挑战.71 5.2 编程模型 .72 5.2.1 编程模型概念.72 5.3 Styx 架构.74 5.3.1 组件.74 5.3.2 事务执行流水线.75 5.4 排序与函数执行 .76 5.4.1 事务排序 .76 5.4.2 调用图发现 .77 5.4.3 函数执行确认 .77

5.5 提交事务 .78 5.5.1 无锁提交阶段 .79 5.5.2 基于锁的提交阶段 .79 5.5.3 调用图缓存 .79 5.5.4 通过确定性提前回复提交结果.80 5.6 容错.81

5.6.1 Incremental Snapshots & Recovery 81
5.6.2 Sequencer Recovery 82
5.6.3 Exactly-Once Processing 83
5.6.4 Exactly-Once Output 84
5.6.5 Addressing Non-Deterministic Functions. 84
5.7 Evaluation 85
5.7.1 设置.85 5.7.2 延迟与吞吐量 .87 5.7.3 可扩展性.89 5.7.4 容错性评估.89 5.8 相关工作.91 5.9 未来工作 .92 5.10 结论.92 5.11 Styx 实践 .92 5.11.1 演示概述 .93

6 状态 Styx 中的迁移：迈向无服务器事务函数 97
6.1 Styx 中的状态迁移.98 6.2 状态迁移方法.100 6.2.1 停止与重启 .100 6.2.2 在线.101 6.3 维护保证 .103 6.4 Styx 上的状态迁移实验.103 6.5 相关工作.106

7 Discussion & Conclusion 107
7.1 Main Findings 107
7.2 Limitations. 109
7.3 Future Directions. 111

参考文献 113

wledgments 135

Curriculum Vitæ 137

List of Publications 139

SIKS Dissertation Series 141

摘要

Web 应用几乎支撑着数字化社会的方方面面，从娱乐、网上购物、假期规划与预订、在线游戏，到通信、工作和社会互动。然而，在现代云环境中构建可扩展且一致的 Web 应用，需要在多个领域具备广泛而多样的专业知识，例如云计算、软件开发、分布式系统和数据库系统，以及领域知识。这些要求使得此类应用的开发只有少数才华出众的人才能胜任，而只有大型企业才能雇用他们。本论文旨在通过识别并应对三项关键挑战，使云应用的开发和维护得以普及：*i)* 云应用的可编程性；*ii*) 具备容错保证的高性能可串行化事务；以及*iii*) 无服务器语义。为应对这些挑战，我们创建了 Stateflow——一种高级、面向对象、易于使用的编程模型；它与 Styx 协同运行。Styx 是一种新颖的确定性数据流引擎，可提供高性能可串行化事务和无服务器语义。

在研究如何普及可扩展云应用这一挑战时，我们发现这类应用与流式数据流执行模型背后的原则十分相似。在第 2 章中，我们重点介绍流式数据流处理与当前最先进水平的事件驱动微服务架构之间的相似之处，并为构建理想的云应用程序运行时指明方向。为验证这一假设，我们改造了一个现有数据流系统，使其支持事务型云应用，并据此创建了 T-Statefun，详见第 3 章。当时，最合适的候选方案似乎是 Apache Flink Statefun，这是一种有状态函数即服务系统（ SFaaS ）；我们为其添加了基于协调器函数的事务支持。通过 T-Statefun，我们证明了数据流系统可以通过 SFaaS API 支持事务型云应用。此外，开发过程还帮助我们发现了两个重大问题： *i)* 编程颇具挑战，尤其是在加入协调器函数之后；以及 *ii)* 由于状态与处理相分离，且事务协议效率低下，T-Statefun 的性能不足。

在本论文中，为了解决可编程性问题，我们在第4 章介绍了 Stateflow。这是一种便于用户使用的编程模型，软件开发者可以采用成熟的面向对象编程风格编写代码，无需编写样板代码，而 Stateflow 会将其转换为基于有状态数据流图的中间表示。在 Stateflow 的实验过程中，我们验证了第3章中发现的消息传递和状态方面的低效问题，以及 Stateflow 支持的其他后端缺乏事务支持的问题。因此，在第5 章中，我们详细介绍了 Styx 的设计。Styx 是一种分布式流式数据流系统，通过高级、标准的 Python 编程模型支持具有可串行化隔离保证的多分区确定性事务，并免除了事务失败管理。我们的设计选择和新颖算法使 Styx 在所有测试工作负载中的吞吐量至少比最先进的系统高一个数量级。

Styx 证明，可以构建一种高性能的 SFaaS 系统，在提供事务和容错保证的同时，采用直观且只需极少样板代码的编程模型。在此基础上，我们扩展了 Styx，使其能够动态、高效地适应不断变化的工作负载。为此，第6章探讨了 Styx 如何以事务方式迁移状态。鉴于 Styx 将应用状态保存在内存中，这项能力是实现弹性所必需的。

我们通过总结关键发现、反思所作贡献、批判性地审视所提出方法的局限性，并思考其更广泛的伦理与社会影响，为本论文作出结论。此外，基于我们在创建 Stateflow 编程模型和 Styx 运行时过程中获得的洞见，我们阐述了该领域面临的新挑战与未来方向。

摘要

Web 应用程序几乎支持我们高度数字化社会的各个方面：从娱乐和在线购物，到假期规划、电子游戏、通信、工作和社交互动。它们在我们的日常生活中发挥着至关重要的作用。然而，在现代云环境中构建此类应用程序，需要在多个领域具备深入且多样化的专业知识，例如云计算、软件开发、分布式系统、数据库系统以及特定领域的知识。这些要求使得此类应用程序的开发仅对数量有限的极具才华的个人，或有能力吸引这类人才的大型企业而言可行。在本论文中，我们旨在通过识别并应对三个重要挑战，实现云应用程序开发与维护的民主化：*i)* 云应用程序的可编程性；*ii)*具备容错能力的高性能串行事务；以及 *iii)* 无服务器语义。为此，我们开发了 Stateflow——一种高级、面向对象且易于使用的编程模型，与 Styx 协同工作。Styx 是一种创新的确定性数据流引擎，支持高性能串行事务和无服务器语义。

在探索如何让可扩展的云应用程序普及化这一挑战时，我们发现它们与流式数据流执行模型的原则高度契合。在第 2 章中，我们强调了流式数据流处理与当前最先进水平的事件驱动微服务架构之间的相似之处，并勾勒出通往理想云应用程序运行时的路径。为检验这一假设，我们开发了 T-Statefun，详见第 3 章：我们通过改造现有数据流系统来支持事务型云应用程序。当时，Apache Flink Statefun——一个有状态函数即服务（SFaaS）系统——似乎是最佳选择，我们通过协调函数为其添加了事务支持。借助 T-Statefun，我们证明了数据流系统能够通过 SFaaS API 支持事务型云应用程序。此外，其开发过程还帮助我们发现了两个重要问题：*i)* 该系统难以编程，尤其是在添加协调函数之后；*ii)* 由于状态与处理相分离，且事务协议效率低下，T-Statefun 的性能不佳。

Om het programmeerprobleem aan te pakken, introduceren we in Chapter 4 Stateflow, een gebruiksvriendelijk programmeermodel waarbij ontwikkelaars software schrijven in een vertrouwde objectgeoriënteerde stijl, zonder enige boilerplate-code. Stateflow vertaalt dit vervolgens naar een intermediaire representatie gebaseerd op toestandsgebaseerde dataflow-grafen. Tijdens het experimenteren met Stateflow bevestigden we de inefficiënties uit Chapter 3 met betrekking tot messaging, toestand en het gebrek aan transactionele ondersteuning in de andere ondersteunde backends. Daarom presenteren we in Chapter 5 het ontwerp van Styx, een gedistribueerd dataflow-systeem dat multi-partitie determinis-

tische transacties ondersteunt met seriële isolatiegaranties, via een hoog-niveau Python programmeermodel dat het afhandelen van mislukte transacties overbodig maakt. Onze ontwerpkeuzes en nieuwe algoritmes stelden Styx in staat om de meest geavanceerde systemen van dat moment met minstens een orde van grootte te overtreffen qua doorvoer in alle geteste workloads.

Styx toonde aan dat het mogelijk is om een hoog-performant SFaaS-systeem te bouwen dat transactionele en fouttolerante garanties biedt, terwijl het een intuïtief programmeermodel met minimale boilerplate behoudt. Voortbouwend op deze basis was de volgende stap om Styx uit te breiden met de mogelijkheid om zich dynamisch en efficiënt aan te passen aan wisselende workloads. In Chapter 6 verkennen we hoe Styx toestand transactioneel kan migreren — een noodzakelijke eigenschap voor elasticiteit, aangezien Styx applicatietoestand in het geheugen houdt.

最后，我们将共同讨论论文的主要发现，并批判性地反思这些贡献。我们将分析所提方法的局限性，并思考其更广泛的伦理和社会影响。最后，基于我们在开发 Stateflow 编程模型和 Styx 运行时期间获得的见解，我们将提出该领域未来研究面临的新挑战和方向。

Περίληψη

Οι διαδικτυακές εφαρμογές υποστηρίζουν σχεδόν κάθε πτυχή της έντονα ψηφιοποι-ημένης κοινωνίας μ
ας, από την ψυχαγωγία και τις αγορές στο διαδίκτυο, μέχρι τονπρογραμματισμ
ό και την κράτηση διακοπών, τα διαδικτυακά παιχνίδια, την επικοι-νωνία,
την εργασία και την κοινωνική αλληλεπίδραση, παίζοντας κρίσιμο ρόλο στηνκαθημερινότητά μας. Ω
στόσο, η ανάπτυξη τέτοιων εφαρμογών μεγάλης κλίμακας α-
παιτεί εκτενή και πολυδιάστατη τεχνογνωσία σε πολλούς τομείς, όπως το υπολογιστικόνέφος,
η ανάπτυξη λογισμικού, τα κατανεμημένα συστήματα και τα συστήματα βάσεωνδεδομένων,
καθώς και εξειδίκευση στον εκάστοτε επιχειρησιακό τομέα. Αυτές οι απαι-
τήσεις καθιστούν την ανάπτυξη τέτοιων εφαρμογών εφικτή μόνο από λίγα εξαιρετικάταλαντούχα άτομ
α ή από μεγάλες εταιρείες που έχουν τη δυνατότητα να προσλάβουντέτοιο προσωπικό.
Σε αυτή τη διδακτορική διατριβή, στοχεύουμε στη δημοκρατικοπο-
ίηση της ανάπτυξης και της συντήρησης τέτοιων εφαρμογών στο νέφος, εντοπίζοντας
και επιλύοντας τρεις βασικές προκλήσεις: i) την προγραμματισιμότητα των εφαρμογώννέφους, ii)
τις υψηλής απόδοσης σειριοποιήσιμες συναλλαγές βάσεων δεδομένων με εγ-γυήσεις ανοχής σε σφάλματα,
και iii) την εκτέλεση των εφαρμογών σε αρχιτεκτονικήχωρίς διακομιστή (无服务器). Για να τις αντιμ
ετωπίσουμε, δημιουργήσαμε το Stateflo- w, ένα υψηλού επιπέδου αφαίρεσης, αντικειμενοστρεφές,
εύχρηστο προγραμματιστικόμοντέλο. Το προγραμματιστικόμοντέλο επιτρέπει την ανάπτυξη εφαρμ
ογών που εκτελο-ύνται από το Styx, μια καινοτόμα ντετερμινιστική μηχανή επεξεργασίας ροών δεδομένων
που παρέχει συναλλαγές με σειριοποιήσιμη εγγύηση απόδοσης.

Κατά τη μελέτη της πρόκλησης της δημοκρατικοποίησης των εφαρμογών νέφους,διαπιστώσα
με ότι αυτές ταιριάζουν στενά με τις αρχές του μοντέλου εκτέλεσης ροώνδεδομένων （流式数据流）。
Στο Κεφάλαιο 2, επισημαίνουμε τις ομοιότητες μεταξύτης επεξεργασίας ροών δεδομ
ένων και των σύγχρονων αρχιτεκτονικώνμικροϋπηρεσι-ών που βασίζονται σε γεγονότα （事件驱动）
και σκιαγραφούμε το ιδανικό περιβάλλονεκτέλεσης για εφαρμογές νέφους。 Για να επαληθεύσουμ
ε την υπόθεσήμας, δημιουργήσα- με το T-Statefun, όπως παρουσιάζεται στο Κεφάλαιο 3, προσαρ
μόζοντας ένα υπάρχονσύστημα επεξεργασίας ροών δεδομ
ένων ώστε να υποστηρίζει συναλλακτικές εφαρμογέςνέφους。 Εκείνη τη χρονική περίοδο,
το καταλληλότερο σύστημα φάνηκε να είναι το Apache Flink Statefun, ένα σύστημα που προσομ
οιάζει μια υπηρεσία εκτέλεσης συναρτήσεωνμε δεδομένα κατάστασης στο υπολογιστικό νέφος 函数即服务（SFaaS）, στο οποίο προσθέσαμε υποστήριξη για συναλλαγές μ
έσω συντονιστικών συναρτήσεων（协调器函数）。 Με το T-Statefun, δείξαμε ότι ένα σύστημ
α επεξεργασίας ρο-ών δεδομένων μπορεί να υποστηρίξει συναλλακτικές εφαρμογές νέφους μ
έσω ενός APIτύπου SFaaS。 Επιπλέον, η ανάπτυξή τουμας βοήθησε να εντοπίσουμε δύο σημαντικά
προβλήματα: i) ήταν δύσκολο στον προγραμματισμό, ιδιαίτερα μετά την προσθήκη των
συντονιστικών συναρτήσεων, και ii) λόγω του διαχωρισμού κατάστασης και επεξεργα-σίας και ενός
μη αποδοτικού πρωτοκόλλου συναλλαγών, το T-Statefun υστερούσε σεαπόδοση。

Για την αντιμετώπιση του προβλήματος προγραμματισιμότητας, στο Κεφάλαιο 4παρουσιάζουμε το Stateflow, ένα φιλικό προς τον χρήστη προγραμματιστικό μοντέλο,στο οποίο οι προγραμματιστές λογισμικού γράφουν σε καθιερωμένο αντικειμενοστρεφέςστυλ προγραμματισμού, χωρίς *καθόλου* 样板代码 κώδικα, ενώ το Stateflow μετατρέπειαυτόματα τον κώδικα σε ενδιάμεση αναπαράσταση βασισμένη σε ροές δεδομένων με κα-τάσταση. Κατά την πειραματική χρήση του Stateflow, επιβεβαιώσαμε τις αναποτελεσμα-τικότητες που εντοπίστηκαν στο Κεφάλαιο 3, αναφορικάμε τη διαχείρισημηνυμάτων,την κατάσταση, ή την απουσία υποστήριξης συναλλαγών στα υπόλοιπα υποσυστήματατου Stateflow. Έτσι, στο Κεφάλαιο 5 παρουσιάζουμε όλες τις λεπτομέρειες του σχε-διασμού τουStyx, ενός κατανεμημένου συστήματος ροής δεδομένων που υποστηρίζειπολυ-κατατμημένες, ντετερμινιστικές συναλλαγέςμε εγγυήσεις σειριοποιήσιμης απο- μόνωσης μέσω ενός υψηλού επιπέδου προγραμματιστικού μοντέλου βασισμένου στηνPython, το οποίο απαλλάσσει τον προγραμματιστή από τη διαχείριση αποτυχημένωνσυναλλαγών. Οι σχεδιαστικέςμας επιλογές και οι καινοτόμοι αλγόριθμοι που ανα-πτύξαμε, επέτρεψαν στο Styx να ξεπεράσει τα πιο προηγμένα συστήματα της εποχήςτου, τουλάχιστον κατάμία τάξη μεγέθους υψηλότερη απόδοση σε όλες τις δοκιμασμένεςπεριπτώσεις φόρτου.

Το Styx απέδειξε ότι είναι εφικτή η ανάπτυξη ενός υψηλής απόδοσης SFaaS συ-στήματος που προσφέρει εγγυήσεις συναλλακτικότητας και ανοχής σε σφάλματα, διατηρώντας ταυτόχρονα ένα διαισθητικό προγραμματιστικόμοντέλο με ελάχιστο απαιτο-ύμενο σκελετό κώδικα λογισμικού στην υλοποίηση των εφαρμογών. Βασιζόμενοι σε αυτότο θεμέλιο, το επόμενο βήμα ήταν η επέκταση του Styx με τη δυνατότητα να προσαρμόζε-ται δυναμικά και αποδοτικά σε μεταβαλλόμενα φορτία εργασίας. Για να το επιτύχουμεαυτό, στο Κεφάλαιο 6 διερευνούμε πώς το Styx μπορεί ναμεταφέρει την κατάστασημε συναλλακτικό τρόπο, μια αναγκαία δυνατότητα για ελαστικότητα, δεδομένου ότι τοStyx διατηρεί την κατάσταση των εφαρμογών στημνήμη.

Ολοκληρώνουμε αυτή τη διατριβή συνοψίζοντας τα βασικά ευρήματα και αναλογιζόμενοι τις συνεισφορές της, εξετάζοντας κριτικά τους περιορισμούς των προτεινόμενων μεθόδων και λαμβάνοντας υπόψη τις ευρύτερες ηθικές και κοινωνικές τους επιπτώσεις. Επιπλέον, βασισμένοι στις εμπειρίες μας από τη δημιουργία του προγραμματιστικού μοντέλου Stateflow και της πλατφόρμας εκτέλεσης Styx, σκιαγραφούμε τις νέες προκλήσεις και τις μελλοντικές κατευθύνσεις στον τομέα.

1

引言

The primary objective of this thesis is to democratize the development lifecycle of large-scale cloud applications. At present, only very few people have expertise in cloud application development, infrastructure, distributed systems, and data management combined. This thesis argues that to enable anyone to code such applications, an easy-to-use distributed programming model and a computing system to serve it are required.

This chapter introduces the fundamental concepts of contemporary large-scale cloud applications and summarizes the contributions of this thesis. Section 1.1 presents the transition from on-premise servers to modern cloud offerings. Following that, Section 1.2 lays out the fundamentals of scalable cloud applications, and Section 1.3 completes the fundamentals with a dive into the different aspects of serverless technology and this thesis interpretation. Sections 1.4 to 1.7 highlight the main research questions, this thesis's contributions, the publications included, and a visual outline of the thesis document.

1

Rapid technological advancements, driven by large-scale cloud applications such as social networks, e-commerce platforms, multimedia streaming services, and AI-driven tools, have fundamentally reshaped how we interact with digital services and data. These applications handle massive volumes of data and requests, ensuring seamless user experiences globally with high availability and minimal downtime. However, building and maintaining such systems is an inherently complex task, often reserved for individuals with extensive technical expertise or large corporations with significant resources.

One of the key challenges in developing these applications lies in their scale, where reliability, performance, and scalability are critical factors. These systems must manage thousands of concurrent users, integrate with diverse data sources, and perform under unpredictable loads, all the while maintaining low latency and fault tolerance. As a result, the design and development of these cloud-based applications demand specialized skills in distributed systems, data management, cloud architectures, and scalable infrastructure.

Architecturally, the journey to this level of sophistication has been gradual but profound. In the early days of software development, monolithic applications were the norm, self-contained systems with tightly integrated components [1]. While monolithic architectures offered simplicity in development, they became difficult to scale and maintain as applications grew. To address these limitations, the industry transitioned toward microservice architectures, where different functionalities are decoupled into independently deployable services [2]. This architectural shift enabled teams to scale, update, and manage services more efficiently.

乍看之下，微服务似乎是替代单体应用并迁移到云端的显而易见的选择。然而，微服务舍弃了单体应用近五十年来一直享有的一项重要优势：状态管理、故障管理和状态一致性原本都由数据库系统负责。如今的微服务架构不再具备数据库管理系统（DBMS）曾经提供的便利，而是将状态管理、服务消息传递和协调整合进应用逻辑中。从数据库领域的角度来看，微服务架构模式让人想起很久以前的情形： [3]，当时，开发人员通过实现临时的应用层事务来确保数据库一致性。

例如，在购物车应用中，要完成结账，我们首先需要确保所选商品库存充足，并预留这些商品；然后在发货前收取款项。在微服务范式中，每个服务（购物车、库存、支付）都有自己的 API、数据库和应用逻辑，并通过 API 调用与其他服务通信。微服务的主要问题在于，原子性（即更新库存并 收取订单款项，或取消这两项操作）以及工作流步骤之间的状态一致性（即库存数量应反映已成功支付的订单）都必须在应用代码中实现。

Unsurprisingly, the easy-to-code Function-as-a-Service (FaaS) [4–6] paradigm embraces the same general architecture pattern as microservices: stateless application, stateful database, and communication via messages or external storage. An orchestration layer on top of FaaS allows composing more complex workflows to build service-oriented applications. However, workflow orchestrators solve only part of the problem. For atomicity and consistency, developers adopt the SAGA pattern or the Two-Phase Commit (2PC) [7]

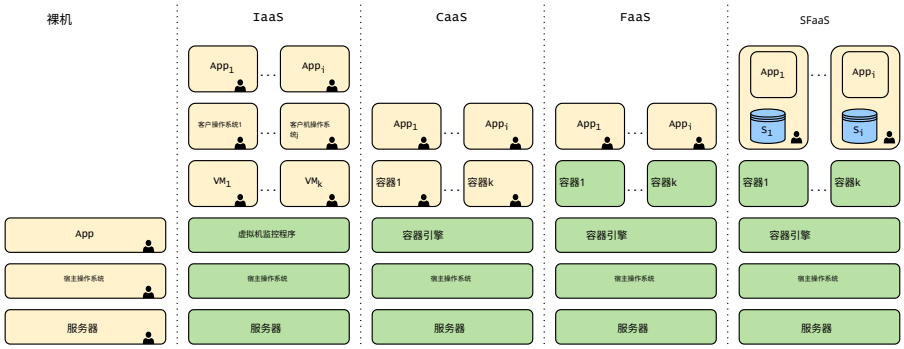


Figure 1.1: Evolution of cloud computing abstractions and the human involvement required

protocol. For applications requiring *transactional* state consistency across services [2], important challenges remain open, including data consistency across service calls and exactly-once guarantees.

By studying these challenges, we identify that the event-driven microservice and orchestrated FaaS architectures inherently form stateful streaming dataflow graphs with partitioned state co-located with the application logic. In short, this architectural pattern is the same as that followed by streaming dataflow systems such as Apache Flink [8] and Spark Streaming [9]. However, modern dataflow systems are primarily built for streaming analytics and do not have an API or important features like transactional support necessary to create general-purpose cloud applications.

This thesis addresses these critical gaps by extending the Stateful Function-as-a-Service (SFaaS) [10–15] paradigm with serializable transactional guarantees, local state management with state migration for elasticity, and fault-tolerant execution. Our contributions significantly simplify the development of scalable, consistent, and reliable cloud applications, democratizing access to sophisticated distributed system features without requiring developers to manage underlying complexity explicitly.

1.1 From the Metal to the Clouds

Database and distributed systems have experienced significant shifts in deployment models over the last decades, evolving from tightly coupled bare metal hardware/software to highly abstracted cloud-based environments, as illustrated in Figure 1.1. Initially, web applications and database systems operated directly on physical servers, providing predictable performance and resource allocation, but with limited scalability and significant operational overhead [16]. Administrators were tasked with manually managing infrastructure components, dealing with hardware failures, and optimizing performance at the physical level. This manual approach often led to downtime during maintenance and limited agility in responding to evolving business requirements.

Virtual Machines (2000s). The introduction of Virtual Machines [17–19] (VMs), as an Infrastructure-as-a-Service (IaaS) solution, marked a key transition. VMs abstract physical

1

hardware, enabling multiple isolated operating systems and database instances to coexist on a single physical host. Virtualization improved resource utilization, simplified infrastructure management, and allowed easier scaling through VM replication and migration. Both on-premise virtualization and cloud providers became commonplace, democratizing access to flexible infrastructure and reducing administrative burdens. However, VMs introduced new challenges, such as performance overhead due to hypervisor abstraction, complexities in VM management, and slower startup times.

容器化（2010 年代）。容器化技术的兴起，[20, 21], 尤其是 Docker 和 Kubernetes，代表了另一项重大进步。容器将应用程序及其依赖项封装在轻量、可移植的单元中，相较于虚拟机，可减少开销。容器化环境的优势包括更快的启动时间、更简便的版本管理，以及更精简的部署流水线。Kubernetes 等容器编排器进一步带来了自动扩缩容和自我修复能力。此外，容器增强了开发、测试和生产环境之间的一致性。这些解决方案称为容器即服务（CaaS）。

无服务器（2020 年代）。最近，无服务器计算通过彻底抽象基础设施管理，改变了整个格局。无服务器架构使数据库系统能够动态、透明地扩展，根据需求波动进行响应，而无需明确预配资源。DynamoDB、[22],Aurora [23], 或 Firestore [24] 等数据库体现了这一趋势，它们能够自动独立扩展计算资源和存储。此外，无服务器计算消除了管理底层基础设施的需要，降低了开发人员和企业的入门门槛，使他们能够更加专注于应用逻辑。

Regarding application logic, serverless computing promotes a shift towards a stateless execution model, exemplified by Function-as-a-Service (FaaS) offerings such as AWS Lambda [4], Azure Functions [6], or Google Cloud Functions [5]. In FaaS, developers break down applications into fine-grained, event-driven functions that can be triggered by external events such as HTTP requests or asynchronous messages. These functions are stateless, with each invocation operating independently and relying on external storage to persist state. This paradigm encourages modularity and fine-grained scalability, as each function can be deployed and scaled independently. However, FaaS introduces new challenges in managing state, coordinating execution across functions, and reasoning about correctness.

To overcome the limitations of FaaS, the Stateful Function-as-a-Service (SFaaS) paradigm has emerged. SFaaS allows functions to maintain state across invocations, enabling richer application semantics without compromising the benefits of serverless infrastructure. In this model, functions can encapsulate state locally or access stateful abstractions through tightly integrated state management systems. Examples include systems like Apache Flink Statefun [25] and Azure Durable Functions [26], which provide mechanisms for long-lived workflows, stateful coordination, and reliable function orchestration. By bridging the gap between stateless scalability and stateful logic, SFaaS makes a step toward making serverless computing viable for general-purpose cloud applications. Furthermore, alongside SFaaS, (Virtual) Actors [27] are a closely related paradigm that addresses the same core challenge: enabling stateful, long-lived interactions in cloud applications while preserving the benefits of serverless infrastructure. Both paradigms aim to abstract away the complexities of distributed state management, fault tolerance, and scalability from the developer, allowing

them to focus on business logic.

This Thesis. In this thesis, we extend SFaaS, the latest serverless paradigm, with serializable transactional guarantees, local state support with state migration support for elasticity, and fault tolerance (Chapters 4 to 6). Making SFaaS closer to being feature-complete for scalable cloud applications.

1.2 Scalable Cloud Application Aspects

Building scalable cloud applications involves navigating the design space of system and programmability tradeoffs. This section highlights key aspects that influence how modern cloud applications are developed and operated at scale. We begin by examining cloud runtimes and programming models, which define how developers express application logic and how systems execute it across distributed infrastructure. We then discuss the importance of transactional guarantees for preserving application correctness. Next, we cover high availability and fault tolerance mechanisms that ensure services remain responsive and resilient despite failures. Finally, adaptivity is addressed, enabling systems to adjust to workload changes and infrastructure constraints dynamically.

1.2.1 Cloud Runtimes & Programming Models

分布式系统的编程模型一直是长期以来的研究主题 [28–32]。在开发云应用的语境下，编程模型和运行时抽象会极大地影响底层系统的设计，反之亦然。撰写本文时，软件工程师最常采用的方法是使用微服务框架（例如 Java Spring [33], 、Python Flask [34]）。除微服务之外，还有一些新兴的云应用编程模型，包括：*i)* Actor（例如 Akka [35], 、Orleans [36]）和*ii)* 有状态函数（例如 Flink Statefun [25], 、Azure Durable Functions [10]），它们在系统设计、抽象以及为开发者提供的保证方面都存在显著差异。

微服务。 为发挥并行处理和松耦合的优势，最常见的方法是按功能将应用逻辑和状态划分为彼此独立的组件，并让它们通过同步或异步消息进行通信 [37], ，这些组件称为微服务。微服务框架提供库和工具，帮助开发者构建微服务。所提供的库可能支持对象关系映射，以便进行数据库交互；支持通过 REST 或消息传递实现服务通信；还提供重试/撤销功能以确保正确性。使用此类框架构建的每个微服务通常都会采用多线程应用服务器。如果某个微服务是有状态的，与其配对的数据库会根据配置的隔离级别处理数据一致性。然而，在全局微服务部署层面，无法提供一致性保证，因为各个数据库彼此独立，无法再利用单个分布式一致性数据库的一致性保证；因此，开发者需要自行实现这些保证（例如临时事务 [38]），这增加了系统的复杂性。

Actors. The actor model is a programming model for concurrent and parallel computation in distributed systems [39]. An actor models a sequential process that performs transformations on the local state based on incoming asynchronous messages. Actor systems achieve concurrency by pipelining and dynamic actor creation [39]. Traditional actor frameworks

1

allow developers to program systems using low-level primitives, such as actor IDs and prescriptions of their physical locations.

虚拟参与者 [36] 是传统 Actor 模型的扩展，在无需开发者处理集群中的参与者分配/调度、生命周期管理、显式创建和销毁参与者实例，以及故障透明性的情况下，提供位置透明性。虚拟参与者已在 Orleans [36] 和 Dapr [40]等流行的分布式应用框架中实现。

云函数。 随着无服务器计算的兴起 [41],，一种名为函数即服务（FaaS）的新型云范式 [10, 42] 逐渐流行。在函数即服务（FaaS）中，开发者将应用程序构建为函数集合。函数执行由外部事件或其他函数的调用触发，从而支持函数工作流组合。

最初，函数即服务（FaaS）主要面向输入/输出和通信量较小到中等的工作负载，因此不鼓励为单个函数内部或跨函数的操作提供数据模型和一致性保证 [11, 13]。

由于这些局限性，人们越来越关注将函数即服务（FaaS）范式扩展到需要频繁访问状态并具备一定一致性保证的应用程序，这类范式称为有状态函数即服务（SFaaS） [11–13, 15, 43]。在有状态函数即服务（SFaaS）中，开发者通过组合函数来编写程序，并使用键值接口访问应用状态。除共享状态接口外，其编程、执行和部署模型与虚拟参与者相似。

有状态数据流。数据流模型规定，应用程序表示为数据流图。这涉及将程序分解为相互独立的处理单元，并将这些处理单元组织成有向无环图（DAG）。处理单元（节点）通过消息流（边）交换数据。数据流主要作为分析型批次处理和流处理系统的编程模型，例如 Spark [44] 和 Flink [8]。在这些系统中，处理单元被定义为算子，用于执行有状态操作（例如连接、聚合）或无状态操作（例如映射、过滤）。消息流可以分区，并分配给不同的并发算子实例。有状态算子通常不共享状态，从而避免并发问题并提升并行性。

然而，数据流模型在用于事务型云应用时存在两个主要问题。首先，数据流系统通常使用函数式编程风格的数据流 API 进行编程，这要求开发者重写云应用，使其符合事件驱动数据流模型。尽管许多云应用可以适配这一范式，但这需要开发者接受大量培训并投入相当多的精力。其次，在数据流之上实现事务并非易事，尤其是实现跨越多个服务且提供可串行化保证的事务 [14, 43, 45, 46]。

This Thesis. In this thesis, we utilize the dataflow execution model, address the transactionality issues (Chapters 3 and 5), and provide a stateful entity domain-specific language for ease of programming that is close to the Virtual Actor programming model (Chapter 4).

1.2.2 Transactional Guarantees

While developing cloud applications, maintaining correctness is essential. In traditional database settings, this usually refers to ACID transactional guarantees [1]. *Atomicity* ensures that either all operations within a transaction succeed or have no effect on the

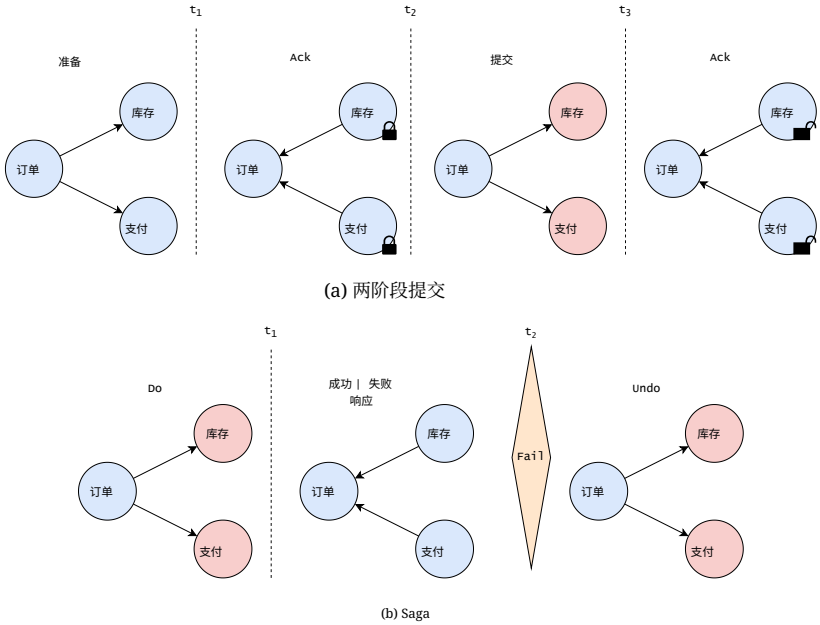


图 1.2：结账场景中两阶段提交与 Saga 事务协议的比较，该场景跨越三个微服务（订单、库存和支付）。会产生状态变更的阶段以红色标记。在此示例中，两阶段提交需要四个时间槽才能提交，而 Saga 只需要两个。

state. *Consistency* guarantees that a transaction moves the database from one valid state to another, preserving integrity constraints. *Isolation* ensures that concurrently executing transactions do not interfere with each other. *Durability* ensures that once a transaction commits, its effects are permanent, even in the case of system crashes.

虽然原子性和持久性相对直观，但一致性和隔离性有不同的形式。近年来，研究人员提出了可串行化性 [47, 48]。后 NoSQL 系统提供的隔离级别也体现了这一理念，例如谷歌的 Spanner [49] 以及最近的 CockroachDB [50]。支持可串行化性的主要理由是，即使在非常复杂的分布式部署中，工程师也能够在存在状态不一致的情况下推理正确性 [47, 48]。在分布式环境中保证可串行化性的标准方式是两阶段提交协议 [1] （如图1.2a所示）。在第一阶段，事务管理器通过死锁检测机制（即等待-死亡或伤害-等待）进行准备，并锁定所请求的键，或返回失败。一旦所有参与方都确认已持有锁，事务便可以提交并解锁这些键。

The counterargument against serializability is the difficulty of providing such a strong guarantee and maintaining high performance, especially when long-running transactions are present in the workload. The most prominent technique for eventual consistency without isolation is the Saga pattern, as shown in Figure 1.2b. A Saga decomposes a transaction into a sequence of local transactions (steps), each performed by a different service or process. If one of the local steps fails, the system executes compensating actions

1

to undo the preceding successful steps. This approach allows for coordination without distributed locking [51, 52] or global consensus [53, 54].

随着云应用的规模和复杂性不断增长，对数据正确性提供强有力保证的需求也日益凸显。然而，目前观察到，许多开发人员忽视了事务的必要性，并尝试创建名为 [38, 47] 临时事务的自定义解决方案。这些事务是由开发人员协调、嵌入应用代码中的数据库操作序列，而非通过传统数据库事务或 ORM 不变式验证来表达。临时事务试图通过手动并发控制来模拟隔离，但往往需要在正确性和性能之间做出权衡。

本论文。上述事实进一步印证了我们关于普及可扩展云应用的主张，也表明我们需要一种简单的编程模型 [55] （第4章）以及 Styx [15] ，以保证可串行化 ACID 事务，且无需开发人员参与（第5章）。

1.2.3 高可用性与容错

可靠系统的两项关键要求是高可用性和容错 [1, 56,57], ，这对于电信、金融服务、交通运输和医疗保健等关键任务型应用尤为重要。这些系统需要持续运行，停机时间接近于零。云服务提供商向软件工程师提供的典型服务级别协议（SLA）规定，每年不可用时间不超过 5 分钟，这对应于所谓的“五个九”（99.999%）可用性。

To narrow the scope, we will focus on higher-level techniques that database or distributed systems employ to give such guarantees. For high availability, the primary mechanism is replication (i.e., maintaining copies of the database across multiple nodes or data centers). In practice, most systems implement failover mechanisms; for example, automatic switching to a standby replica when the active/primary node fails. Moreover, load balancing incoming requests and partitioning/sharding the database state help distribute the load more evenly so that more nodes share the load at peak times.

Database systems use a plethora of mechanisms to maintain fault tolerance. Some of them are: *i)* periodically capturing state to enable rollback and recovery after failures (snapshotting/checkpointing [58]), *ii)* logging state mutations before applying them to ensure durability (write-ahead logging[59]), *iii)* transaction mechanisms, as mentioned in Section 1.2.2, *iv)* requiring agreement from a majority before committing operations in the presence of replicas (quorum and voting [53, 54, 60, 61]).

This Thesis. The primary contribution of this thesis, Styx (Chapter 5), utilizes periodic coordinated snapshots [58, 62] and a heartbeat failure detection mechanism for fault tolerance. Furthermore, we use minimal write-ahead logging to maintain determinism throughout the system. Although not directly addressed, high availability is straightforward to implement using async replication, as discussed in Section 7.3.

1.2.4 Adaptivity

Adaptivity is a critical requirement for scalable cloud systems. As workloads evolve due to user demand, temporal access patterns, or changes in system topology, systems must con-

tinuously adapt to maintain performance and correctness. Adaptivity manifests in several forms, most notably in the system's ability to autoscale [63–66] and, in the case of stateful systems (e.g., stream processing engines, databases, or stateful microservices/functions), migrate state across machines [67–70].

Autoscaling. Adaptive systems offer autoscaling to automatically adjust the number of compute instances based on workload metrics such as CPU usage, request latency, or custom application-level signals. Autoscaling is relatively straightforward for stateless systems by spawning or removing instances, whereas for stateful components, it requires careful orchestration to ensure seamless continuity and consistency. For example, spawning a new instance might require retrieving its state from a shared store or other replicas, potentially introducing latency or consistency issues if not handled properly. Furthermore, fine-grained function-level autoscaling in serverless systems must balance responsiveness and the overhead of startup, warmup, or reconfiguration [71].

State Migration. To support adaptivity in stateful systems, state migration is essential. It enables redistributing the application state across nodes, whether to balance load, scale up-/down, or recover from failures. State migration often involves moving partitions/shards or individual keys in data-intensive systems. Effective migration mechanisms must minimize downtime, ensure no data loss, and avoid violating consistency or transactional guarantees.

本论文。 Styx（第 5章）提供状态迁移功能（第 6章），同时确保零停机时间、一致性和事务保证。由于自动扩缩容与此正交，因此留待未来工作处理。

1.3 什么是无服务器?

无服务器计算 [72–77]是一种云计算范式，其特点是将基础设施管理抽象化、实现自动扩缩容，并允许按粒度计费、按使用量付费。通过将运维事务从开发者身上移开，无服务器计算带来了更简单的开发模式、更高效的资源利用率以及更灵活的经济实践。在第 1.1节中，我们讨论了无服务器计算对数据管理和可扩展云应用的影响；本节将探讨无服务器计算的三个维度：开发者体验、底层系统考量以及相关的经济权衡。

1.3.1 开发者视角

从开发者的角度来看，无服务器计算通过抽象掉基础设施相关事务，让开发者能够专注于业务逻辑 [72, 76]，从而大幅简化云应用的开发。无服务器应用由事件驱动的无状态函数组成，这些函数由 HTTP 请求、数据库更新或消息等外部事件触发。资源配置、容器管理和扩展所涉及的复杂性完全由云服务提供商负责管理。

This paradigm dramatically reduces developers' effort, facilitating rapid prototyping, faster iteration, and shorter deployment cycles. Nevertheless, developers must adapt to constraints intrinsic to serverless architectures, including function statelessness, ephemeral execution environments, and runtime limitations [72, 75]. Despite these limitations, the overall productivity gains and reduced operational complexity are considerable.

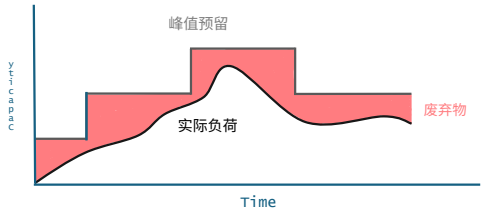


Figure 1.3: In a typical scenario, operators provision machines based on the expected peak, which results in much waste. With Serverless solutions, operators do not need to provision any machines, which matches actual load and reduces waste.

1.3.2 Systems' Considerations

Serverless computing demands sophisticated automation and efficient resource management from the underlying execution systems. Cloud platforms dynamically allocate resources and scale functions in response to changing demand with minimal intervention. Elasticity is achieved through mechanisms such as rapid cold-start initialization, function reuse, and fine-grained resource isolation across tenants [72, 76]. However, these properties impose significant system challenges.

一项关键挑战是降低冷启动延迟，即初始化函数执行引擎时产生的延迟，这会显著影响延迟敏感型应用程序的性能。此外，由于固有的无状态性，函数必须与外部存储服务交互，因此高效处理状态与数据局部性也更加复杂 [75, 76]。另一项挑战是，在负载多变且难以预测的情况下实现性能可预测性，这需要采用复杂的调度和编排策略。

1.3.3 经济学

无服务器计算通过使资源成本与消耗直接挂钩，从根本上改变了云经济学。与传统的基础设施即服务（IaaS）或平台即服务（PaaS）模式不同，在这些模式下，开发者需要预先分配资源，并承担闲置基础设施的成本；无服务器模式仅对应用程序实际使用的执行时间和资源收费 [72, 76]。

这种精细化的计费模式显著降低了间歇性或不可预测工作负载的成本，使资源得以高效利用，而无需预先投入资金。因此，组织可以更自由地进行实验、迭代和创新，而不必承担资源过度配置带来的经济风险 [75]。由此，无服务器计算让更多人能够使用云规模计算，使小型团队、初创公司以及使用模式高度多变的应用程序受益。

However, this consumption-based model may become economically unfavorable for sustained or predictable workloads. Traditional reserved resource models may offer superior cost efficiency in scenarios with constant or highly predictable resource usage. Therefore, enterprises must evaluate workload characteristics carefully to determine the optimal economic strategy when adopting serverless computing [75, 76].

1.3.4 This Thesis Interpretation

This thesis primarily focuses on the developer's perspective, adding transactional guarantees, fault tolerance, stateful functions, and determinism. Our approach provides lower latency, higher throughput, and stronger consistency guarantees while maintaining ease of use. We explore elasticity with state migration mechanisms, and although critical to serverless adoption, we leave the economics out of scope.

1.4 Main Research Questions

In this thesis, the primary research question is to investigate the possibility of creating a system and framework with an easy-to-use programming model that allows non-expert developers to write complex cloud applications free of concurrency or machine failure considerations while maintaining high performance.

To that end, we have split this into five research questions:

> **RQ-1:** What would be the optimal substrate for a system serving complex cloud applications?

在第 2 章中，我们通过首先从软件开发者的视角阐述通往理想云运行时的路径来回答 **RQ-1**。此外，我们指出了现代云架构（即事件驱动微服务）与有状态数据流图之间的相似之处。为验证我们的推论，我们提出以下研究问题。

> **研究问题二：** 是否可以为此目的使用现有的 SFaaS 数据流系统？如果可以，它有哪些局限性？

为回答 **研究问题二**，如第 3 章所示，我们首先尝试使用 Flink-Statefun [25], 这一成熟的 SFaaS 系统，并为其添加所有必需的事务编排。尽管它的性能优于最先进水平，但仍存在一些局限性。由于其实现的死锁预防机制，可串行化协议在高竞争场景中的吞吐量较低。Flink-Statefun 的架构会将状态传输到远程处理工作器，而不是在流处理器内部进行处理，从而增加了延迟。此外，它的 API 包含大量样板代码，令非专业开发者难以使用。这引出了第三个研究问题：

> **RQ-3:** Can we design a domain-specific language that runs on top of all stream processing systems, providing a simple, easy-to-use object-oriented API?

Answering **RQ-3** is essential for the democratization of the development of large-scale cloud applications. In Chapter 4, we present *Stateflow*, a programming model and intermediate representation (IR) that compiles imperative, transactional object-oriented applications into distributed dataflow graphs and executes them on existing dataflow systems. Instead of designing an external Domain-Specific Language (DSL) for our needs, we opted for an internal DSL embedded in Python, which is already popular for cloud programming and is easy to use. Specifically, a given Python program is first compiled into an IR, an enriched stateful dataflow graph independent of the target execution engine. The choice of execution engine is entirely independent of the application layer, which allows switching

1

to different ones with no changes to the application code. However, all current systems had limitations, leading to poor performance, and our fourth research question:

研究问题4： 我们能否构建一个系统，使开发者无需具备分布式系统方面的专业知识，也能编写事务性、数据密集型云应用?

为此，在第 5 章中，我们展示了如何构建 Styx：一种新颖的、用于 SFaaS 的基于数据流的运行时。它在确保恰好一次执行的同时，支持任意函数编排，并提供端到端可串行化保证；此外，它借鉴了确定性数据库的概念，以避免代价高昂的两阶段提交。我们的工作源于两个关键观察。首先，现代流式数据流系统（如 Apache Flink） [8] 通过向开发者隐藏故障，保证精确一次处理 [8, 78, 79] 。然而，这些系统无法用于执行微服务等云应用，更无法保证事务性 SFaaS 编排。其次，能够避免昂贵的两阶段提交调用的确定性数据库协议 [80, 81] 并非为复杂的函数编排和调用图而设计，因此不能直接满足 SFaaS 的需求。尽管 Styx 满足了高性能要求，但其资源使用方面缺乏灵活性，因此引出了第五个也是最后一个研究问题：

研究问题5： 我们能否赋予 Styx 弹性特性，例如状态迁移，使其成为无服务器系统?

过去几年中，围绕流式数据流系统的动态重新配置 [82–84] 以及状态迁 [67–69] 移的研究取得了许多进展。这些进展对于在状态与计算共置的情况下实现无服务器弹性至关重要，并使数据流图能够作为 无服务器有状态云应用的执行模型；相关内容将在第 6 章介绍。

1.5 贡献

本论文的主要贡献及其开源代码成果总结如下：

1. 我们刻画了云应用运行时，并为现代环境下的“理想”运行时指明了方向。我们推断，现代事件驱动微服务范式与数据流引擎的基本原理高度契合，并指出数据流引擎可以充当云应用运行时。（第2章）

2. 为验证我们在第 2 章中的推断，我们探讨了将现有 SFaaS 系统 Apache Flink Statefun 改造为满足云应用程序运行时需求的可能性。为此，我们实现了协调器函数，在 Apache Flink Statefun 中提供事务支持，并提供不同的一致性保证。我们将新系统命名为 T-Statefun[1] ；其性能比当前最先进的事务型函数即服务系统高出一个数量级。其性能也至少比分布式 OLTP 数据库高 1.5 倍。（第3章）

3. In Chapter 3, we addressed all the functional requirements of a dataflow runtime that serves scalable cloud applications. Next, we created an easy-to-use domain-specific

[1] https://github.com/delftdata/flink-statefun-transactions

language called Stateflow[2]. Stateflow takes object-oriented code, where an object is a stateful entity, and transforms it into the dataflow execution model. We have proven the ease of integration of Stateflow with existing dataflow systems and the minimal overhead it adds to those. (Chapter 4)

4. We created a new distributed dataflow engine, Styx[3], that serves as a runtime for scalable cloud applications. Based on lessons learned from Chapters 3 and 4, we ensure that the state is local to the dataflow operator and allows for direct addressing of operators since other systems had to go through the ingress if they wanted to respond to another operator. These design changes required a few algorithmic changes and optimizations that enabled Styx to outperform the T-Statefun (Chapter 3) and the state of the art by an order of magnitude while providing serializable transactional guarantees and coarse-grained fault tolerance. (Chapter 5)

5. We extended Styx with state-of-the-art state migration capabilities, a step towards Styx becoming elastic, leading to it becoming serverless. Our experiments show minimal impact of the migrating actions on Styx's throughput and latency. (Chapter 6)

1.6 Thesis Origins

The main body of the thesis consists of five main chapters based on the research papers listed below:

Chapter 2 is based on the following publication:

K. Psarakis, G. Christodoulou, M. Fragkoulis, and A. Katsifodimos. Transactional Cloud Applications Go with the (Data)Flow, CIDR'25 [46].

Chapter 3 is based on the following publications:

🏆[4] *M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed transactions on serverless stateful functions, ACM DEBS'21* [43].

M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Transactions Across Serverless Functions Leveraging Stateful Dataflows. In Elsevier's Information Systems, Volume 108, September 2022 [14].

Chapter 4 is based on the following publications:

K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows (Abstr.), CIDR'23 [85].

K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows (Vision),

[2] https://github.com/delftdata/stateflow
[3] https://github.com/delftdata/styx
[4] The trophy icon indicates that the paper won the best paper award

1

EDBT'24 [55].

Chapter 5 is based on the following publication:

K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. Styx: Transactional Stateful Functions on Streaming Dataflows, ACM SIGMOD'25 [15].

K. Psarakis, O. Mraz, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. Styx in Action: Transactional Cloud Applications Made Easy (Demo), VLDB'25 [86].

Chapter 6 is based on the following publication:

K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. State Migration in Styx: Towards Serverless Transactional Functions (Under Review).

Additionally, this dissertation benefits from the following research papers:

R. Laigner, G. Christodoulou, K. Psarakis, A. Katsifodimos, Y. Zhou. Transactional Cloud Applications: Status Quo, Challenges, and Opportunities (Tutorial), ACM SIGMOD'25 [87].

G. Siachamis, K. Psarakis, M. Fragkoulis, A. van Deursen, P. Carbone, A. Katsifodimos. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows, IEEE ICDE'24 [62].

G. Siachamis, G. Christodoulou, K. Psarakis, M. Fragkoulis, A. van Deursen and A. Katsifodimos. Evaluating Stream Processing Autoscalers, ACM DEBS'24 [66].

1.7 Visual Outline

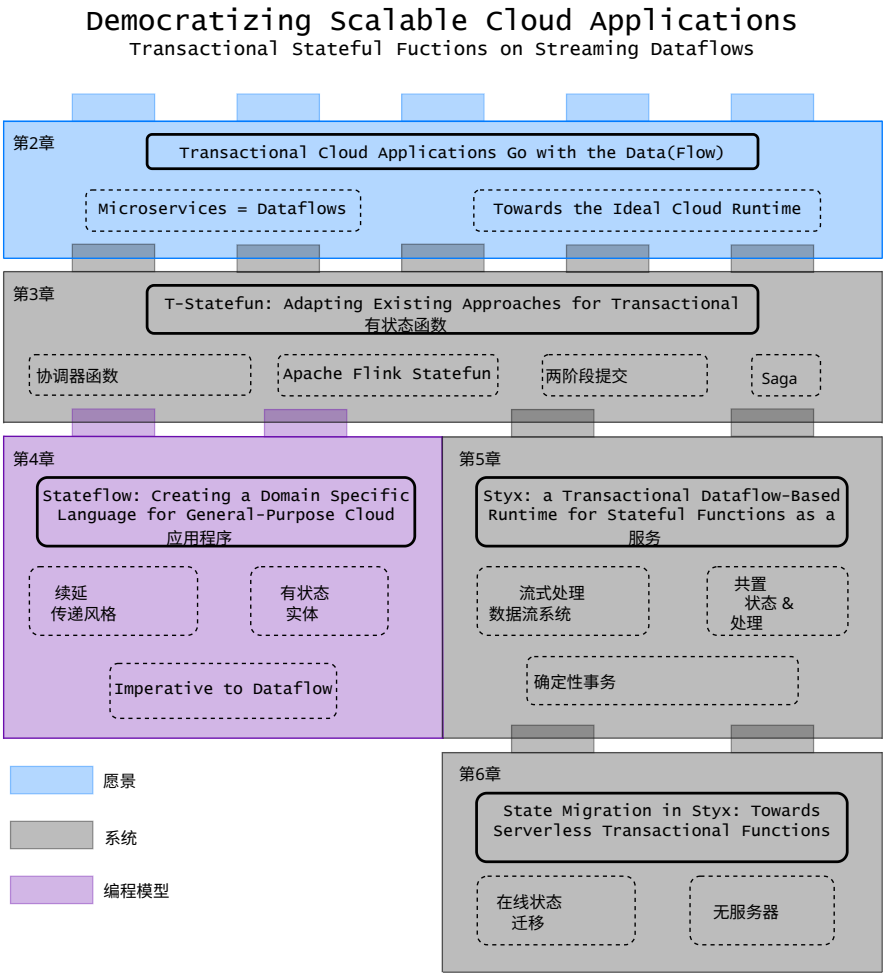


Figure 1.4: Visual outline of this thesis with the chapters and main ideas put into them.

2

Transactional Cloud Applications Go with the Data(Flow)

Traditional monolithic applications are migrated to the cloud, typically using a microservice-like architecture. Although this migration offers significant benefits, such as scalability and development agility, it also leaves behind the transactional guarantees that database systems have provided to monolithic applications for decades. In the cloud era, developers build transactional, fault-tolerant distributed applications by explicitly implementing transaction protocols at the application level.

This chapter presents the main argument of this thesis and outlines our approach: the principles underlying the streaming dataflow execution model and deterministic transactional protocols provide a powerful and suitable substrate for executing transactional cloud applications.

Parts of this chapter have been published in:
K. Psarakis, G. Christodoulou, M. Fragkoulis, and A. Katsifodimos. Transactional Cloud Applications Go with the (Data)Flow, CIDR'25 [46].

Over the last decades, enterprises have migrated applications such as order management systems, banking systems, game-backend services, and supply chain management to the cloud. The transition from monolithic applications follows an architectural pattern that favors a stateless application layer supported by a stateful database layer. All the stateless and stateful components communicate with each other via REST calls or message queues. Microservice architectures are well-known instances of this pattern.

2

乍一看，微服务显然是替代单体应用并迁移到云端的理想候选方案。微服务提供了代码模块化、可扩展性和开发敏捷性。然而，微服务失去了单体应用近五十年来一直享有的一项重要优势：状态管理、故障管理和状态一致性都由数据库系统负责。如今的微服务架构不再享有这些数据库管理系统的便利，而是将状态管理、服务消息传递和协调与应用逻辑交织在一起 [88]。从数据库领域的角度来看，微服务架构模式类似于很久以前所描述的情形 [3], : 当时，开发人员实现临时的应用层事务，以确保数据库一致性。更糟糕的是，在分布式云环境中管理通信和状态会增加复杂性。

例如，在购物车应用中，要完成结账，我们首先需要确保所选产品库存充足，然后在发运产品之前收到付款。在微服务范式中，每个服务（购物车、库存、支付）都有自己的 API、数据库和应用逻辑，并通过 API 调用与其他服务通信。微服务的主要问题在于，原子性（即更新库存并且收到订单付款，或者取消这两个操作）以及工作流之间的状态一致性（即库存数量应反映已成功支付的订单）都必须在应用代码中实现。同样，函数即服务（FaaS）遵循与微服务相同的总体架构模式：无状态应用、外部数据库和基于消息的通信。FaaS 之上的编排层支持复杂工作流的组合，从而构建面向服务的应用。

However, orchestrators [26, 89, 90] solve only part of the problem, namely the atomicity of a workflow's execution. Moreover, achieving atomicity typically requires developers to handcraft compensating actions to roll back changes correctly using the SAGA pattern [91]. To address these concerns, a line of research [12, 13] proposes FaaS systems for workflow orchestration with transactional guarantees at the expense of performance and high-level programming primitives. For applications requiring low-latency transaction execution and state consistency across services [2], important challenges remain open.

In this chapter, we first identify the limitations and shortcomings of microservice-like architectures for implementing transactional applications and then motivate the need for dedicated runtimes to support transactional cloud applications. We argue that to remove transaction- and failure-handling code from the application level, we need to address complex orchestration, service calls, and state management in a *holistic* manner at the system level, i.e., via a dedicated runtime. During the last few years, we have been developing a runtime for transactional applications called Styx (Chapter 5) [15]. Styx automatically partitions state, parallelizes function execution, and enables arbitrary transactional workflows with low latency. Most importantly, Styx's programming model (Chapter 4) [55] allows for application development that resembles a single-node application/monolith

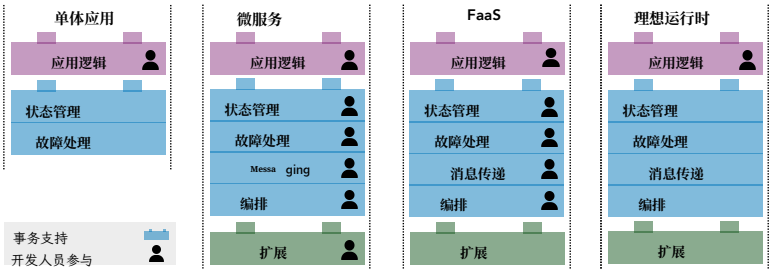


Figure 2.1: In monolithic applications, developers focused on application logic while a transactional database handled state management and failure recovery. In distributed cloud applications, development involves more challenges (e.g., failures, exactly-once messaging, and orchestration for atomicity and scalability). The ideal runtime should offer the same state consistency and ease of programming as monoliths, with improved scalability, without developer involvement.

while transparently handling the serializable execution of massively parallel workflows in the cloud.

Our work is in line with recent research, such as Orleans [27], DBOS [92], Hydroflow [93], and SSMSs [94]. Contrary to these systems, our work adopts the streaming dataflow execution model while exposing an object-oriented/actor-like programming model on top [55] and guarantees serializability *across* services.

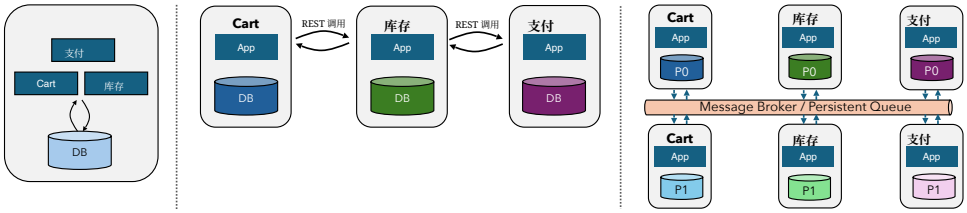
To summarize, in this chapter, we make the following contributions:

- We analyze the shortcomings of modern cloud applications by exemplifying issues with current architectures and requirements for future systems (Section 2.1).
- We provide arguments on the suitability of the stateful streaming dataflow paradigm for transactional cloud applications (Section 2.2).
- We introduce a novel approach that combines ideas from deterministic databases, dataflow systems, and serverless architectures (Section 2.3).

2.1 From Monoliths to Microservices

As illustrated in Figure 2.1, developers in monolithic architectures were primarily responsible for the application logic. At the same time, with the adoption of microservices, they need to deal with messaging and failures (Section 2.1.1), state management and orchestration (Section 2.1.2), and scaling techniques (Section 2.1.3). Interestingly, in Figure 2.2, we observe that these aspects are not orthogonal. The conversion to a partitioned, event-driven architecture (Figure 2.2b to Figure 2.2c) requires state migration, coordination, and fault-tolerance.

Figure 2.2 depicts the process of breaking down a monolithic application (Figure 2.2a) into three microservices, each with its own database (Figure 2.2b). In the microservice architecture, direct access to a single database and DBMS-based transactions are no longer possible. Instead, the microservices split functionality and maintain their own database. Each service's database is partitioned to scale out, as shown in (Figure 2.2c). REST API calls are also transformed into messages that asynchronously trigger those calls.



a) 单体应用 b) 微服务 c) Partitioned, event-driven architecture

Figure 2.2: Three-step process of converting a monolith to a scalable, low-latency service architecture.

Microservices Implement Dataflows. A critical observation is that the architecture depicted in Figure 2.2c closely resembles a streaming dataflow graph with the partitioned state co-located with the application logic. While we elaborate on this in Section 2.2.2, in short, this architectural pattern is the same pattern that is followed by streaming dataflow systems such as Apache Flink [8] and Spark Streaming [95].

2.1.1 Messaging, Idempotency & Consistency

Traditional monoliths achieved workflow execution atomicity (e.g., a shopping cart checkout) by combining state mutations across subsystems (cart, payment, stock) in a single transaction. If the transaction fails, the database rolls back to the previous state, and the application retries the checkout.

Idempotency in Services. To achieve the same effect, a stateful service or function must be idempotent, meaning that calling the service multiple times should have the same effect on the global state of an application as calling it exactly once. Considering that various issues can arise [96] when two services communicate (such as network failures, rescaling, or service restarts), currently ensuring idempotency works as follows: the sender service generates an *idempotency-key*[1] that is persisted in the state of the sender, right before the call is performed. Suppose the sender sends a message twice (e.g., because of an intermittent network issue or a failure). In that case, the *idempotency-key* must be recognized and safely ignored by the receiving service. It is important to note here that idempotency cannot be achieved without *persisting* the idempotency-key to durable storage (e.g., a database) in the *same local transaction* as the one that mutates the state of the receiver. At the moment, *idempotency-keys* are managed by the developers, adding to the complexity of developing cloud applications.

2.1.2 Transactions & Orchestration

Serializability in Services. Multiple works advocate that serializable guarantees are preferred [47, 48]. This is also reflected in the offered isolation levels of post-NoSQL systems such as Google's Spanner [49] and, more recently, CockroachDB [50], which all provide serializability. Serializability has been highly important in monolithic applications, but in distributed service deployments, it is virtually impossible to reason about correctness in the presence of state inconsistencies [48]. Transactional service architectures must

[1] https://datatracker.ietf.org/doc/html/draft-idempotency-header-00

address message delivery guarantees.

Saga 模式与两阶段提交。 多年来，业界一直采用 Saga 模式和由实现 XA 事务的事务协调器编排的两阶段提交协议 [52]，作为应对此挑战的常见方案。然而，这两种方案都存在显著的缺点。实现 Saga 模式需要管理补偿操作的执行，以撤销工作流失败后产生的部分状态效应，而其提供的一致性级别是最终一致性。另一种方案是采用两阶段锁定配合两阶段提交协议，以阻塞事务所涉及的服务编排的进展为代价，提供原子性和隔离性。我们需要一种全新的方式来构建云应用程序，使其支持跨越应用程序多个组件的事务工作流。

编排器。 目前，有多种商业编排器可用于执行 Saga。这些编排器只确保原子性：它们保证给定的一系列服务调用最终能够完成。虽然我们确实看到了编排器在分析应用中的价值（例如 Apache Airflow [97],、AWS Step Functions [90]），但编排器并不适用于事务型应用，因为它们都对所编排的函数/服务的状态一无所知 。

2.1.3 Application (Re-)Scaling

Scaling microservices requires scaling the stateless business logic and the state management system that serves the stateless part of an application. Scaling stateless services is relatively straightforward: one needs to rescale the application logic instance, assuming that the database behind the stateless instance can handle the new load. However, when optimizing for latency, the database is partitioned and preferably co-located with the application logic. In that case, rescaling an application becomes a hurdle: the database has to migrate state and possibly keep replicas. Soon enough, application developers re-implement some version of database state migration and rescaling [98] protocol.

While current FaaS cloud offerings do allow for stateless functions to scale on demand, they still provide no transaction management primitives that take into account service orchestrations and state consistency issues during the rescaling process. An ideal runtime should be able to perform the rescaling of applications without forcing operations teams and developers to perform rescaling by hand while keeping the state across services transactionally consistent.

2.2 Streaming Dataflows to the Rescue

In this section, we highlight the key aspects and advantages of streaming dataflow systems design and argue that they can be extended to encapsulate the primitives required for executing transactional cloud applications consistently and efficiently. Moreover, we argue that combining deterministic databases and dataflow systems can create a runtime that ensures atomicity, consistency, and scalability. Finally, we show how deterministic databases can be extended for SFaaS, where transaction boundaries are unknown, unlike online transaction processing (OLTP).

2.2.1 数据流作为架构抽象

Stateful dataflows are the execution model implemented by virtually all modern stream processors [99]. Streaming systems owe their wide adoption in the last decade to a set of key system design aspects: exactly-once processing, consistent fault tolerance, co-location of state and compute, and data-parallel scale-out architecture. We elaborate on these characteristics below.

精确一次处理。 在一般情况下，消息传递保证从根本上难以处理，其根源是众所周知的拜占庭将军问题。然而，在数据流系统的封闭环境中，精确一次处理是可行的 [8, 78]。原则上，要实现精确一次处理，处理层会记录每条消息所产生的状态效应，网络层确保消息按先进先出顺序传递，而容错层则保证已反映到状态中的消息不会再次处理。值得注意的是，精确一次处理的保证大大简化了编程。流行的流式数据流系统（例如 Apache Flink）的应用程序接口无需编写错误管理代码（例如消息重试，或使用幂等键消除重复项）。

容错。 精确一次处理也延伸到了系统的容错方法。借助适用于流处理系统的 Chandy-Lamport 分布式快照协议 [58] [62, 78]，二者可以优雅地结合起来。这种方法会定期向流式数据流系统中发送称为检查点标记的特殊消息，指示其算子对状态进行快照。由于检查点标记与普通的数据相关消息共用同一通道，因此它们会强制形成全局顺序，从而为系统状态创建一个一致割。发生故障时，系统可以自动回滚到其分布式状态的最新检查点，并从该处恢复处理，前提是输入来自可重放源，例如 Apache Kafka [100]。这种容错方法可确保系统状态在发生故障时保持一致。

Co-location of State with Compute. Streaming dataflow systems have demonstrated their capacity to process millions of events per second [8]. One main design decision that enables this level of sustainable performance is that the system's operators maintain the state of their computations in their local memory space. The state is periodically snapshotted to persistent storage, securing the progress of continuous computations against failures. Notably, this coarse-grained approach bears a low overhead to the system's regular operation.

Data-parallel Scale-out Architecture. Continuing from the previous point, the system's architecture enables high-throughput at scale. Each operator in the logical dataflow graph is instantiated as several operator instances deployed in distributed nodes. Each instance holds a partition of the operator's state, enabling input data to be distributed and processed in parallel across instances.

2.2.2 Dataflows for Transactional Applications

The aforementioned advantages of streaming dataflow systems do not apply to transactional cloud applications. To begin with, typical transactional workloads in the cloud manifest as workflows of functions that arbitrarily call one another. This computation pattern

is markedly different from analytics functions that populate the operators of streaming systems. Second, streaming dataflow systems lack support for transactions as prescribed in the database literature [3]. Finally, the development of workflows of functions entails a programming model that can convey transactional semantics, form workflows, and support custom business logic. This programming model departs from the typical way of programming stream-processing jobs as chained functional transformations.

Dataflows for Arbitrary-Workflow Execution. The prime use case for dataflow systems nowadays is streaming analytics, which typically involves executing a chain of standalone functions. By comparison, transactional cloud applications involve arbitrary workflows of functions calling each other. To enable the execution of arbitrary workflows in a dataflow system, we connect operators at the system level such that an operator can directly invoke a computation in another operator. In addition, we allow such nested computations to be executed in parallel. Finally, we devised an approach for identifying the transaction boundaries of a workflow, which we briefly describe next.

Deterministic transactions. Deterministic transactional protocols have two properties that make them coexist harmoniously with dataflow systems. First, given a set of sequenced transactions, a deterministic database [80, 101] will reach the same final state with serializable guarantees despite node failures and possible concurrency issues. This property is essential because it allows a deterministic transactional protocol to align with a dataflow system without changing the stream processor's checkpointing mechanism.

其次，与两阶段提交不同，后者在发生故障时需要回滚，而确定性数据库协议 [80, 81]是“只向前”的：一旦确定了 [80]事务批次的加锁顺序或读写集 [81]，事务便会执行并反映到数据库状态中，无需回滚更改。确定性数据库与数据流执行模型之间的这种契合，是在数据流系统之上支持确定性事务协议的主要动因。

不过，在流式数据流系统中支持确定性事务并非易事，存在两项主要挑战。我们在第2.3节介绍的原型系统中解决了这些挑战。首要挑战是确定事务边界。在确定性数据库中，无需处理这一问题，因为每个事务都封装在一个单线程函数中，该函数可以跨分区执行远程读写 [80, 81]。然而，在 SFaaS 中，调用任意函数以访问远程分区十分常见，因为这使开发者能够同时利用微服务架构中广泛采用的关注点分离原则 [2] 和代码模块化。因此，为了确定事务工作流的边界，我们引入了一种用于统计工作流中嵌套函数调用的计数方案。该方案还支持调用远程算子和处理循环，并会在所有函数调用完成后发出信号，表示工作流执行已终止。

第二个挑战是决定何时将数据提交到持久化存储并向用户回复。传统上，事务系统只有在 *i*) 请求的事务已提交到持久、可靠的状态，或 *ii*)预写日志已刷新并复制后，才能向客户端作出响应。在数据流系统的范畴内，这需要完成一次快照，从而导致难以接受的延迟。然而，确定性事务协议会在工作节点之间执行一系列约定好的事务；发生故障后，系统会以完全相同的效果重新运行这些事务。这种确定性使得可以提前回复提交结果：客户端可以在持久化快照

已存储。

编程模型。目前，数据流系统只能通过函数式编程风格的数据流 API 进行编程：开发者需要重写给定的云应用，使其符合事件驱动的数据流范式。尽管可以在这一范式下重写许多应用程序，但这需要程序员接受大量培训并投入相当多的精力。因此，我们引入了一种面向对象编程抽象，将函数封装为类 Actor 实体。我们在第 2.3.1 节完整介绍这一编程模型。我们认为，这种编程模型适用于开发微服务等事务型云应用。

2.3 The Stateflow/Styx Approach

Styx (Chapter 5) [15] is a transactional distributed dataflow system that executes workflows of stateful functions with serializable guarantees. Styx adopts Stateflow (Chapter 4) [55] as a higher-level programming abstraction, enabling users to code in a pure object-oriented style without state management or fault tolerance considerations. In this section, we briefly describe the programming model (Section 2.3.1) and underlying system (Section 2.3.2).

2.3.1 Programming Model

The Stateflow/Styx framework provides developers with two levels of abstraction: a high-level actor-like programming interface based on Stateflow [55] and a lower-level dataflow API [15].

高层。用户可以使用 Python 面向对象代码编写事务型云应用，其中实体是具有唯一键的对象，并包含可改变实体状态的类函数（类似于 Actor 编程）。此外，当一个实体调用另一个实体的函数时，Stateflow 会自动在数据流图中创建一条边。我们将在 [55]中介绍 Stateflow 的工作原理，以及它如何利用续延传递风格编程，将不同实体之间的调用转换为分布式数据流图。

底层。Styx 遵循数据流系统的算子 API（例如 Apache Flink [8]）。在 Styx 中，流式算子可以根据分区方案持有多个实体，也可以基于作用于整个算子的函数（支持范围查询），或基于实体本身的函数（支持点查询）来处理这些实体。为实现跨算子通信，开发者可以使用 Styx 的 API 调用远程算子函数。

2.3.2 Styx 运行时

Styx 采用典型的工作节点/协调器架构。该架构还配备了消息系统（例如 Apache Kafka），用于将输入传播至 Styx，包括在发生故障后重放未处理消息。协调器负责将用户定义的数据流图部署到工作节点，监控集群健康状况并收集有用的指标，以及在发生故障时触发容错流水线。

The workers are responsible for a subset of the dataflow graph's operator state partitions, which are 1-to-1 aligned with the partitions of the replayable input source, say Apache Kafka. First, each worker ingests client requests through Kafka and sequences them (Styx uses a non-replicated sequencer partitioned per worker). Then it receives a batch of transactions from the sequencer and executes them as coroutines on a single CPU to

improve efficiency. To execute transactions deterministically, Styx extends a deterministic transactional protocol similar to Calvin [80] and Aria [81]. Determinism is required by the dataflow snapshotting mechanism to guarantee the same state mutations after a replay in case of failure. Transactions are executed in parallel across workers, and nested function calls are transparently scheduled for execution by local or remote operators. Finally, Styx's acknowledgment-sharing scheme signals the end of a transaction's execution.

容错。为了从故障中恢复，Styx 依赖可重放输入源，根据记录的偏移量执行确定性消息重放。这种设计确保排序器在恢复后重新创建相同的事务序列，并支持提前回复（在状态提交到持久化存储之前）。最后，Styx 使用对象存储持久化工作节点状态的增量快照。

2.4 相关工作

我们的系统与 Hydroflow [93] 和 DBOS [92]等项目有着相同的设计动机。DBOS 采用以数据库为中心的方法：函数可以转换为数据库中的存储过程（实现状态与处理的共置），也可以在服务器上运行，此时状态需要传输；而工作流则构成具有 ACID 保证的数据库事务。Hydroflow 目前尚不支持端到端事务工作流，主要专注于面向分析的云原生流处理。Cloudburst [11] 在单个有向无环图 (DAG) 工作流内提供因果一致性保证。Netherite [102] 提供恰好一次执行保证和高级编程模型，但无法确保函数间的事务可串行化。Orleans [27]引入了虚拟参与者，使应用程序与底层架构解耦，但不保证恰好一次消息传递。最后，具有可串行化保证的事务型 SFaaS 范式（Beldi [13], 、Boki [12], 和 T-Statefun [43]）确实支持端到端事务工作流，但性能较差，也未能将用户代码与其事务原语解耦。

3

T-Statefun: Adapting Existing Approaches for Transactional Stateful Functions

Chapters 1 and 2 introduced the motivation for supporting general-purpose cloud applications with strong consistency guarantees. This chapter investigates whether an open-source platform can be adapted to support transactions in stateful cloud functions.

Before building a custom system from scratch, we sought to understand whether existing open-source platforms could be adapted to meet our goals. In this chapter, we present T-Statefun, our extension to Apache Flink StateFun, a Stateful Function-as-a-Service (SFaaS) platform built atop a stream processing engine that already offers exactly-once processing guarantees.

T-Statefun introduces two complementary models for transactional coordination across stateful functions: the Saga pattern for eventual consistency and two-phase commit (2PC) for serializability. By implementing both on StateFun's dataflow runtime, we explored how far a general-purpose streaming engine can be stretched to support transactional workflows typically required in microservices and cloud-native applications.

Finally, the limitations we observed with T-Statefun informed the design of our system, Styx (Chapter 5). Thus, this chapter serves as both a feasibility study and a key design probe.

This chapter is based on the following research paper and its extended journal version:
M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed transactions on serverless stateful functions, DEBS '21 [43].
M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Transactions Across Serverless Functions Leveraging Stateful Dataflows. In Elsevier's Information Systems, Volume 108, September 2022 [14].

The idea of democratizing distributed systems programming is not new. Approaches such as Distributed ML [103] and Erlang [104] aim to simplify the programming and deployment of distributed applications. Erlang [104] first introduced the actor model, which Akka [105] implemented later in Scala, offering a low-level programming model. Following that, Virtual Actors [27, 36] try to abstract away the low-level primitives.

无服务器计算 [106] 是一种云计算执行模型，有望简化可扩展云应用的编程、部署和运维。在无服务器模型中，开发团队通过高级 API 上传代码，云平台则负责应用程序的部署和操作。无服务器计算旨在解决云领域的现状，从而大幅推动云的普及；在当前的云环境中，开发团队需要具备分布式系统、数据管理以及云执行模型内部机制方面的技能，才能有效使用云服务。

函数即服务与消息传递。 最突出的无服务器服务是函数即服务（FaaS），用户编写函数，云服务提供商则自动完成部署和运维。然而，FaaS 服务缺乏状态管理支持，也无法跨多个函数执行事务工作流 [75, 107],，而通用云应用程序需要这些能力。此外，目前的 FaaS 方法都无法提供消息传递保证，因此无法支持精确一次处理：即函数针对每条传入消息仅修改一次其状态。

如果系统无法保证精确一次处理，那么调试和处理系统错误（例如机器故障、网络分区或落后节点）的重担就会落在开发者身上 [108]。于是，这些开发者不得不在业务逻辑中“掺入”额外的一致性检查、状态回滚、超时处理或恢复机制等内容 [109]。结果是，应用程序代码的大部分内容并非业务逻辑，而是错误检查、管理和缓解措施 [2]。迟早，在应用层编写分布式系统程序会导致状态一致性问题、漏洞，最终造成严重的服务中断。

一般情况下，消息传递保证从根本上说很难处理，问题的根源是众所周知的拜占庭将军问题 [110]。然而，在数据流系统这个封闭的世界中，精确一次处理是可行的[8, 78, 79] ，因为对于有状态数据流，系统可以全面控制消息传递和状态管理。据我们所知，Apache Flink 的 StateFun [25] 是首个构建在流式数据流系统之上的 FaaS 执行引擎方案，即使面对复杂的故障场景，也能提供精确一次处理保证。不过，StateFun 的方案也可以构建在其他数据流系统之上 [8, 9, 111–113]。

Such dataflow systems can execute stateful functions as follows: incoming events represent function execution requests routed to continuous stateful operators that execute the corresponding functions. With proper, consistent fault tolerance mechanisms [78, 79], state-of-the-art stream processing systems operate at high throughput and low latency. At the same time, they guarantee the correctness of execution even in the presence of failures. As we show in this paper, this set of properties can support *transactions* with minimal involvement from application developers.

Transactional SFaaS. Although there is ongoing work on supporting stateful FaaS (SFaaS) applications that mutate state transactionally, *across* functions, remains an open problem. The only approach addressing distributed transactions in an SFaaS setting is Beldi [13],

which provides fault-tolerant ACID transactions on stateful workflows across functions by logging the functions' operations to a serverless cloud database. Cloudburst [11] with HydroCache [114] provides causal consistency on function workflows forming a DAG by leveraging Anna [115], a key-value store with conflict resolution policies in place. Cloudburst does not provide isolation between DAG workflows.

In contrast with the aforementioned approaches, developer teams in the microservices and cloud applications landscape go to extreme lengths when they need to implement transactional workflows across the boundaries of a single service or function. The most common approach adopted is the Saga pattern [91]. The Saga pattern separates a transaction into sub-transactions that proceed independently with the benefit of improved performance, but at the risk of having to undo or compensate the changes of successful sub-transactions when at least one of the involved sub-transactions fails. In addition, compensating actions can be challenging when concurrent changes are applied to the state because Sagas do not require any means of isolation. For this reason, state consistency needs to be dealt with at the application level. On the other hand, applications that prioritize consistency over performance implement distributed transactions using the two-phase commit protocol. Two-phase commit (TPC) [1] offers ACID, serializable transactions, but imposes blocking operations across functions participating in a transaction, which penalizes performance in return for strict atomicity.

In this chapter, we draw inspiration from best practices in developing microservices and cloud applications and offer developers a programming model that supports both Sagas and distributed transactions with two-phase commit. Our implementation for authoring workflows across stateful functions in FaaS with transactional guarantees is publicly available on GitHub[1]. We implement the two approaches on an open-source stateful FaaS system, Apache Flink [8] StateFun [25], and call our extension T-Statefun.

In summary, this chapter makes the following contributions:

- We argue for implementing transactional workflows on a stateful dataflow engine and outline its advantages.
- We propose a programming model for transactional workflows across stateful serverless functions.
- We implement the two main approaches used by cloud application practitioners to achieve transactional guarantees: two-phase commit and Saga workflows.
- We evaluate two transactional schemes using an extended version of the YCSB benchmark on a cloud infrastructure.
- We compare against the state-of-the-art academic SFaaS proposal that supports serializable transactions and one of the most popular transactional distributed database systems.

[1] https://github.com/delftdata/flink-statefun-transactions

3.1 流式数据流上的事务

Serverless platforms come in different flavors. One breed of SFaaS systems (e.g., Apache Flink StateFun and [107]) is built on top of a stateful streaming dataflow engine. This architecture bears important implications for supporting transactions because of how distribution, state management, and fault tolerance work.

典型流式数据流引擎中分布式组件之间的网络通信通过先进先出网络通道实现，这些通道能够保证精确一次处理并保持传递顺序。在无服务器函数即服务系统中，这一特性免除了处理丢失消息以及为事务工作流中的函数调用实现重试逻辑的需要。消息错误和重试是应用层摩擦和发展成本的重要来源，而这些由底层数据流系统提供。

3

最先进的流处理系统通过定期对系统的分布式状态进行一致性快照来实现精确一次处理保证 [78]。这些快照捕获系统在特定时间点的全局一致状态，并用于在发生故障时恢复系统状态。精确一次意味着，即使发生故障，每个函数执行实例所带来的更改也只会在系统状态中记录一次。对于事务而言，这项能力至关重要，因为事务的故障恢复可以直接依托底层容错机制，无需应用程序付出任何努力或具备相关知识。鉴于代码和工作中很大一部分都用于故障处理，系统层提供的容错和虚拟弹性 [116]能够发挥重要作用。

Furthermore, unlike traditional streaming queries, where the computations are fully encapsulated within the system's operators, it is common to have nondeterministic side effects (typically calls to external services or remote key-value stores) in microservices and cloud applications. However, the traditional fault tolerance mechanisms of streaming dataflow systems were not designed to support non-determinism prevalent in general-purpose applications. Thus, the consistency of applications and the integrity of transactions are at risk when transactions involve nondeterministic operations. Extending the fault tolerance approach of streaming dataflow systems to support nondeterministic computations [79] is an important step towards opening their adoption for executing general-purpose applications. Recent work [11, 93] also recognizes the dataflow model as a key enabler for the SFaaS systems of the future.

In short, we believe that stateful streaming dataflows and the associated research that has been proposed so far[117–119] can alleviate the burden of building rich stateful and transactional applications on top of streaming dataflows. This paper presents a step in this direction.

3.2 Preliminaries

本节首先介绍我们的事务模型（第 3.2 节）。接着，在第 3.2 节中，我们将描述 Apache Flink StateFun 的功能和内部机制，它构成了我们所提出解决方案的基础。最后，在第 3.2 节中，我们将列出 SFaaS 系统要成为本工作的后端所需满足的要求。

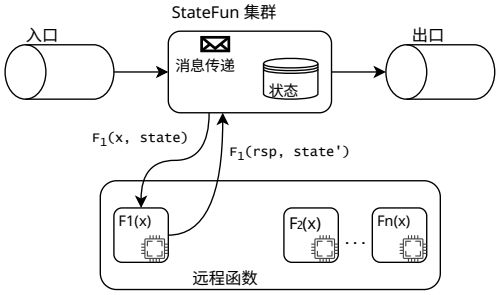


Figure 3.1: Flink StateFun Cluster Architecture

事务模型

在本工作的语境下，事务是对一组有状态函数调用进行的原子执行。更具体地说，本文引入的事务模型考虑的是预先定义的事务。以往工作将此称为单次 [120] 或一次性 [121]事务。我们遵循 H-Store 对 [121]一次性事务的定义：一个函数（查询）的输出不能用作同一事务中后续函数（查询）的输入。由于函数的输出不会被后续函数使用，事务工作流中各函数的执行彼此独立。这简化了系统内的事务协调，同时仍能提供实用的事务模型。广泛使用的数据库服务（例如 Amazon 的 DynamoDB） [122], 支持一次性事务 [120]。在无服务器函数即服务系统中，一次性事务具有显著优势：函数可以使用 Java 或 Python 等通用编程语言实现任意业务逻辑，而不必局限于特定数据库（如 DynamoDB）所支持的 API。因此，这一优势使编程模型具有相当大的灵活性。

Apache Flink StateFun

Apache Flink StateFun[2] 为用户实现有状态云函数提供了一种抽象和运行时。由用户代码实现的有状态函数称为函数类型，用于描述其持有的状态。同一函数类型可以有多个实例并行存在，并由标识符加以区分。每个函数实例都封装自身的状态，并可通过其类型和标识符进行唯一寻址。函数实例可以由其他函数实例调用，也可以通过 Kafka 等入口点调用。函数实例可以产生四种受控副作用：（1）状态更新；（2）函数调用；（3）延迟函数调用；（4）出口消息（例如 Kafka 消息）。StateFun 支持从入口到出口的端到端恰好一次保证，其中包括任何状态更新。

[2] https://flink.apache.org/stateful-functions.html

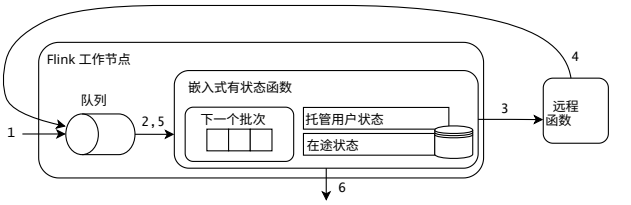


图 3.2：Flink StateFun 的原始通信流程

架构。在图3.1 中，我们展示了 Apache Flink StateFun 的通用系统架构。系统通过入口/出口模式与外界交互（例如，接收/生成 Kafka 消息）。Apache Flink StateFun 集群位于系统核心，由多个管理消息传递和分区状态的工作节点组成，从而支持无状态远程函数。不过，这意味着状态必须随请求一起传输到各个特定函数进行处理。处理完成后，响应和新状态都会返回到 StateFun 集群。这种架构的主要优点是，由于 StateFun 同时管理消息传递和状态，因此比其他架构更容易实现恰好一次语义。

嵌入式函数与远程函数。函数既可以部署在 StateFun 工作进程内部（称为嵌入式函数），也可以部署在 StateFun 集群外部（共置函数和远程函数）。嵌入式函数只是 Flink 中有状态流处理算子之上的一种抽象，因此能够提供精确一次和容错保证。StateFun 通过在流处理图中引入一个环路，支持这些流处理算子之间的动态通信。共置函数和远程函数完全无状态，因为状态持久化在 StateFun 中。本文重点关注远程函数，因为它们可以利用 AWS Lambda 等现有 FaaS 服务，自动扩展计算层。图 3.2 展示了远程函数的工作方式。每个函数实例在 StateFun 集群中都由一个嵌入式有状态函数表示。这个标准化的嵌入式函数负责管理函数实例的状态，并与可部署在任意位置的远程函数通信。图3.2展示了嵌入式有状态函数中持久化的数据，以及远程函数的通信模式。

Function Invocations as Dataflow Messages. Invocations that are sent to a function instance arrive in a queue, as shown in step 1 of Figure 3.2. If the embedded stateful function is ready to process the next invocation, it pulls a message (invocation parameters) from the queue (step 2). When no invocation is being executed at the remote function, the remote function is called. However, if the remote function is busy with a previous function call, the current invocation message is appended to the next batch. Batching is used as an optimization in order to avoid multiple remote calls to external functions at the expense of latency (see Section 3.6.1). Batches are also used to preserve the invocation order and the order of state access (the batch must wait until the state updates caused by the previous batch have been applied), thus ensuring linearizability at the function instance level.

In step 3, the stateless remote function is called through a Protobuf interface that

contains both the (keyed) state required for the remote function to operate and the invocation parameters of the function. The stateless remote function can execute the (batch of) invocations and will be ready to return the updated state back to the Flink worker that made the call. In step 4, the response of the stateless remote function is appended to the queue of incoming messages to the function. The response includes any side effects caused by the invocation(s), including updates to the user-defined state.

处理来自无状态函数的响应（步骤 5）时，会将调用产生的副作用应用于嵌入式函数的状态，从而更新嵌入式有状态函数中的托管状态。如果将调用组成批次，则会将下一批调用发送给远程无状态函数，并截断当前批次。如果没有待处理的批次，则会清除在途状态。最后，将所有发出的函数调用发送到相应函数实例的队列，并将出口消息发送到相应的出口（步骤 6）。

假设与要求

正如我们将在下一节中介绍的，协调器函数依赖底层 SFaaS 系统来记录进行中的事务状态并提供可靠消息传递。为此，底层系统应满足两项要求。

精确一次处理保证。 首先，所有通信都必须可靠，并且必须以精确一次处理保证执行。因此，我们要求底层系统具备容错能力 [8]，以确保发生故障时的事务原子性。这也意味着，即使发生故障，状态在快照/检查点之间也能持久保存。如果我们能够依赖精确一次处理保证、消息重放和错误处理，那么事务协调的大部分工作都可以简化。Flink StateFun 确实提供精确一次处理保证。

可线性化操作。 第二项要求是，任何特定函数实例上的操作都应当是可线性化的，也就是说，这些操作在该实例及其封装的状态上执行时，应当有一个定义明确的顺序。因此，函数调用始终能获取函数实例的正确状态，以实现事务。由于 Flink StateFun 的函数实例为每个函数实例的状态使用单个副本，并且由单个进程以先进先出方式顺序执行该函数实例的函数调用，因此可以确保每个函数实例上的操作都是可线性化的。

3.3 协调器函数与 T-Statefun API

本节介绍有状态协调器函数的概念，并概述我们的方法。我们的方法基于一个简单的观察：由于底层的 SFaaS 系统提供精确一次处理和消息传递保证，从概念上讲，将事务协调器实现为普通的有状态函数会简单得多。基于这一点，我们选择在有状态函数之上实现事务 API，并在表 3.1 中介绍该 API。值得注意的是，要将事务抽象提升到更高的层次 [107, 118]，仍需进一步研究，例如将其作为语法糖。

一个 sta teful coordinator function is a stateful function that preserves state about the

函数	描述
共享协调器函数方法	
send_on_success(type, id, message)	Sends a message to another function instance if the transaction is successful
send_after_on_success(delay, type, id, message)	Sends a delayed message if the transaction is successful
send_egress_on_success(type, egress_message)	Sends a message to an egress if the transaction is successful
send_on_failure(type, id, message)	Sends a message to another function instance if the transaction failed
send_after_on_failure(delay, type, id, message)	Sends a delayed message if the transaction failed
send_egress_on_failure(type, egress_message)	Sends a message to an egress if the transaction failed
Two-phase commit function methods	
tpc_invocation(type, id, message)	Add a function invocation to the transaction
send_on_retryable(type, id, message)	Sends a message if the transaction aborted because of a deadlock
send_after_on_retryable(delay, type, id, message)	Sends a delayed message if the transaction aborted because of a deadlock
send_egress_on_retryable(type, egress_message)	Sends a message to an egress if the transaction aborted because of a deadlock
Saga 函数方法	
saga_invocation_pair(type, id, message, compensating_message)	Add a pair of a message and a compensating message to the transaction
普通函数	
FunctionInvocationException	Raised to fail the function invocation

Table 3.1: Coordinator functions' Python API.

```
1  def serializable_transfer(context, message: Transfer):
2      subtract_credit = SubtractCreditMessage(amount=message.amount)
3      context.tpc_invocation("account_function",
4                             message.debtor,
5                             subtract_credit)
6
7      add_credit = AddCreditMessage(amount=message.amount)
8      context.tpc_invocation("account_function",
9                             message.creditor,
10                            add_credit)
```

Example 3.1: Two-phase commit coordinator function.

execution of a given transaction. Coordinator functions have the ability to force other function instances to abort or compensate for the changes they applied.

API Overview. Our coordinator function implements two transaction coordination patterns: two-phase commit and Sagas [91]. A complete example of a coordinator function for two-phase commit and Saga is shown in Listings 3.1 and 3.2, respectively. In short, to coordinate a two-phase commit transaction, the user needs to invoke function instances via *tpc_invocation*, while for a Saga, an invocation *pair* is expected, which consists of the normal transaction invocation and the corresponding compensation invocation to be sent to the same function instance. A Saga invocation pair can be called with *saga_invocation_pair*. An important difference between the behavior of the two schemes is that a failure in a Saga workflow will incur a compensating function call.

两阶段提交。 Example 可串行化_传输 函数（示例 3.1 接收一个语境（即我们扩展 StateFun 以支持事务后所使用的底层语境）和一条消息。该消息的类型为 *Transfer*，包含三个字段：传输的金额、债权人和债务人。消息中提到的数量必须从债务人的账户中扣除，并传输给债权人。为此，假设系统中已注册了一个函数类型 *account_*函数，按照原始 StateFun API，我们需要构造一个对象，其中包含传递给 *account_*函数 的参数，并将该消息推送给事务协调器。具体操作见第 5—7 行：我们将要调用的函数类型以及债务人的标识符提供给 TPC协调器，以构成该函数实例的地址

```
def sagas_transfer(context, message: Transfer):
    subtract_credit = SubtractCreditMessage(amount=message.amount)
    add_credit = AddCreditMessage(amount=message.amount)
    context.saga_invocation_pair("account_function",
                                 message.debtor,
                                 subtract_credit,
                                 add_credit)
    context.saga_invocation_pair("account_function",
                                 message.creditor,
                                 add_credit,
                                 subtract_credit)
```

Example 3.2: Saga coordinator function.

，以及该函数实例的 *SubtractCreditMessage*，该消息将作为参数传递给该函数。随后，我们对债权人执行相同的操作：构造一个 *AddCreditMessage*，并将其传递给函数类型 *account_*函数。简而言之，事务协调器函数实例将确保以可串行化保证调用这两个函数实例。它通过在函数实例之间协调两阶段提交协议，并结合锁定来确保隔离。关于这些方面的更多详细信息，请参见第 3.4节。

Sagas. Similarly to two-phase commit, our API offers the ability to specify Sagas: as seen in Example 3.2, the *saga_invocation_pair* function in line 6 will receive the target function name, the ID of the debtor as well as two messages: the *subtract_credit* and its compensating action *add_credit*. If there is a failure during the execution of *subtract_credit*, our Sagas transaction coordinator will execute the compensating action *add_credit*, which will put back the original credit to the debtor's account. The details on how Sagas are executed are given in Section 3.4.

对普通函数的扩展。为了让两类协调器函数能够在任意函数实例之间执行事务，需要对普通函数进行一些扩展。首先，参与协调事务的函数必须能够显式失败。将故障通知协调器函数后，会导致事务回滚。目前，Flink StateFun 中没有让一次调用失败的概念；函数调用可以只是不执行任何副作用。为了支持显式失败，我们在 StateFun 与远程部署的函数之间的协议中添加了一个包含这些详细信息的字段。从 API 的角度来看，可以通过抛出异常来触发函数故障。粗略来说，函数故障相当于传统数据库术语中，基于函数实例所封装状态而发生的完整性约束违例。其次，需要更改任何批处理机制。TPC 协调器函数确保事务彼此隔离。这意味着，属于此类事务的任何函数调用都不能与其他函数调用一起批处理。第三，应在函数实例层面实现适当的锁定，以确保基于两阶段提交协调器函数的可串行化事务具有隔离性。最后，函数实例应透明地与协调器函数通信，以免给开发人员带来这项工作负担。

3.4 事务工作流

本节将更详细地介绍我们的 Python API，并介绍我们在 Apache Flink StateFun (T-Statefun) 上针对有状态无服务器函数间事务工作流的实施方案。我们的实施方案由协调器函数组成，它们通过两阶段提交来确保分布式可串行化事务，或将 Saga 工作流作为无隔离事务来实现。

3.4.1 协调器函数

协调器函数负责跨普通有状态函数编排事务工作流。为此，协调器函数封装其负责的活动事务工作流的状态，但不保存参与函数执行的状态或自定义用户定义状态。协调器函数可以通过其名称（内部由类型唯一标识）以及在初始化时随机生成的标识符来调用。随后，输入消息将到达协调器的输入队列。如果协调器函数参与正在进行的事务，则该消息将排队，直到正在执行的工作流完成。协调器函数的 Python API 列于表 3.1。

图 3.3 展示了协调器函数与普通函数实例之间的常见通信流程。两阶段提交和 Saga 工作流中的具体通信方式分别在第 3.4.2 节和第 3.4.3 节中介绍。在这两种情况下并非总会发送的消息均以 * 标注。图 3.3 展示了普通函数实例相比图 3.2更加丰富的内部结构。这些扩展是我们为普通函数实现的，使其能够参与事务工作流。

3.4.2 Saga 协调

Saga 协调器函数的编程模型如示例所示3.2 通过一个示例进行说明。表 3.1介绍了该 API。在 Saga 中，开发者负责定义函数调用对，使第二个函数的调用能够补偿第一个函数的调用 [91]。此外，Saga 协调器函数可以根据事务的完成场景（成功或失败）定义副作用（例如，传出的出口消息）。在当前实施中，组成 Saga 的函数调用会并行执行。接下来，我们将介绍图 3.3 中 Saga 特有的消息。

初始化和远程协调器函数调用。 首先，向协调器函数发送一条消息以初始化事务（步骤 1）。从队列中取出该消息以初始化事务（步骤 2）。然后，使用传入的消息调用远程 Saga 协调器函数（步骤 3）。远程函数将 Saga 工作流的定义返回给其嵌入式对应函数（步骤 4）。其中包括事务涉及的函数调用及其补偿调用，以及在成功或失败时要执行的副作用。

Processing the Remote Coordinator Function's Result. When the embedded function processes the result of the remote function (step 5), a random transaction ID is generated, and a map is created holding the addresses of function instances and the result of their

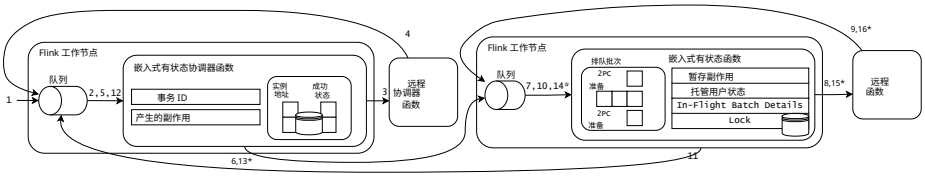


Figure 3.3: Communication flow for transactions in T-Statefun.

execution (at this stage, those are initialized as *null* values). It follows that only one invocation per function instance can be involved in a particular workflow. If multiple invocations of a single function instance are required, this can be solved at the application level by allowing a single message, which combines multiple function invocations, to be sent to the function instance.

调用普通函数。在步骤 6 中，每个参与其中的普通（非协调器）函数实例都会在其输入队列中收到一个函数调用。所有调用会同时发送，因此函数实例可以并行处理。这些函数调用可以被识别为属于某个 Saga 工作流的函数调用。每个 Saga 函数调用都会从队列中取出，然后直接发送给远程函数，或与其他调用一起批处理以提高效率（步骤 7）。由于 Saga 不要求隔离，因此一个函数调用可以与其他调用一起批处理。随后，该调用会被发送到普通远程函数（步骤 8）。处理完成后，函数的响应会被添加到 StateFun 中其嵌入式表示的状态队列中（步骤 9）。在步骤 10 中，嵌入式有状态函数处理无状态远程函数的响应时，会使用正在执行的函数调用元数据中的索引，以及添加到 Protobuf 接口的新列表（即普通函数扩展），来确定 Saga 函数调用的结果状态和相应协调器的地址。如果函数调用失败，则不会产生任何副作用。此后，该函数可以继续处理其他函数调用。

Saga 成功与补偿。 根据 Saga 函数调用的成功状态，系统会向协调器函数发送成功或失败消息（步骤 11）。嵌入式协调器函数处理每个函数调用的成功状态时，会将映射更新为成功或失败状态（步骤 12）。如果某个函数实例失败，则会向所有已成功执行函数调用的函数实例发送相应的补偿操作（步骤 13），并执行故障情况下的副作用处理（步骤 14、15、16）。协调器函数必须等待收到所有函数调用的结果后才能完成。如果任何函数调用失败，协调器函数就会向所有已成功处理调用的函数实例发送补偿消息。请注意，无需向失败的函数实例发送补偿调用，因为这些实例没有产生任何副作用。补偿消息按照常规消息进行处理，仅在任何函数调用失败时才需要。这意味着，如果 Saga 工作流很可能失败，其性能会更差，因为这需要额外的消息传递和处理，开销最多可翻倍。事实上，这正是 Saga 等乐观事务方法所带来的权衡。

3.4.3 两阶段提交协调

示例 3.1中，我们介绍了两阶段提交协调器函数的编程模型；表 3.1 展示了两阶段提交 API 提供的函数。与 Saga 协调器函数类似，两阶段提交协调器函数也可以为任何完成场景定义要执行的副作用。除了成功和失败完成之外，两阶段提交事务还可以以“可重试”状态完成。当事务因死锁而中止时，就会出现这种情况。下文将介绍图 3.3所示的两阶段提交工作流。请注意，工作流的初始化（即步骤 1-5）与 Saga 中的相同，因此这里不再详述。

***PREPARE* 和两阶段锁定增长阶段**。在步骤 6 中，向每个相关函数实例发送其对应的函数调用。此消息可识别为两阶段提交协议的*PREPARE* 消息。当两阶段提交函数调用到达嵌入式有状态常规函数时，如果该函数当前有进行中的调用批次，则此两阶段提交函数调用不会与其他调用一起批处理。相反，两阶段提交函数调用会拆分批次，并将其隔离发送到远程函数，如图3.3所示。这种做法增加了批处理机制的复杂性，因为它现在需要使用批次队列，而不是图3.2所示的仅追加批次。

调用常规远程函数。在步骤 7 和 8 中处理消息（及当前状态）并将其发送到远程函数时，事务 ID 和两阶段提交协调器函数的地址会存储在进行中的调用批次的详细信息中。此时还会锁定函数实例。来自无状态远程函数的响应包含函数调用状态以及任何副作用（步骤 9）。假设无状态远程函数抛出了 *FunctionInvocationException* 。在这种情况下，会丢弃远程函数的响应，向协调器函数实例发送响应以通知其调用失败，并解除常规函数实例的锁定，因为该实例知道事务将被中止。如果函数调用成功，则保留锁定，并向协调器函数实例发送成功响应。随后，状态效应会作为暂存副作用存储在函数实例中（步骤 10）。函数实例处于锁定状态期间到达的任何其他消息都会放入排队批次中。

ABORT & Two-phase Locking Shrinking Phase Upon Failure. The message at step 11 notifies the two-phase commit coordinator function instance whether the function invocation succeeded. If the two-phase commit function instance receives the message that a function invocation failed (step 12), it immediately sends an *ABORT* message to all other function instances and performs the appropriate side effects (step 13), and calls the two-phase lock shrinking phase. After this, the two-phase commit function is done.

COMMIT & Two-phase Locking Shrinking Phase. If the two-phase commit function instance receives the message that a function invocation was successful, it updates the map it keeps of all involved function instances. If all function instances succeed, it sends *COMMIT* messages to all involved function instances and publishes the appropriate side effects (i.e., applies the changes to the embedded function state).

COMMIT/ABORT & Two-phase locking Shrinking Phase. When a function instance receives a *COMMIT* message (step 14), it executes its staged side effects, releases the lock

and continues processing the next request. When a function instance receives an *ABORT* message, it discards its staged changes, releases the lock, and continues processing. Note that a function could also receive the *ABORT* message while the *PREPARE* message is still in the queue or in-flight. In this case, the *PREPARE* message is discarded. Messages 15 and 16 are never sent for two-phase commit transactions.

死锁检测。由于使用了锁，两阶段提交协议容易发生死锁。当两个或更多不同的两阶段提交事务等待其他事务持有的函数实例上的锁时，就可能发生死锁。为处理死锁，我们实现了一种死锁检测机制，说明如下。两阶段提交事务中的所有参与方都可以分布在不同的机器上，而活动事务的状态则封装在不同的协调器函数实例中。因此，我们不希望事务依赖任何集中式组件来处理死锁。我们实现了Chandy-Misra-Haas 算法[123]，它提供了一种简单的方法，可以在分布式方式下检测死锁，而不依赖单个全局协调器。检测到事务中发生死锁时，该事务会立即以可重试事务的形式完成，并向所有相关函数实例发送*ABORT* 消息。收到可重试结果状态后，两阶段提交普通函数可以向自身发送带有相同初始消息（也可以附带一个计数器）的延迟调用，以执行重试。是否这样做由开发者决定，从而使系统能够灵活适用于各种用例。

3.5 T-Statefun 的事务保证

我们采用两阶段锁定协议，因此能够提供可串行化事务。在本节讨论的某些事务场景下，我们的方法可以实现严格可串行化，即事务的处理顺序与它们到达系统的顺序一致。要实现严格可串行化，我们的方法还需要进行扩展。下文将说明，为了支持不同级别的可串行化性，系统中需要做出的各种设计决策或变更。

单分区事务。单线程算子实例会对给定分区的状态执行每项操作。因此，单分区事务必定以串行方式处理。由此也可知，即使以分布式方式执行，单分区事务也能保证严格可串行化。此外，对不同分区进行操作的事务可以实现水平扩展。

多分区事务。一般而言，我们的方法中的事务可以访问多个函数、修改多个状态分区，或两者兼有。由于采用了两阶段锁定，系统可以在同一函数的多个函数和数据分区之间强制保证可串行化。此外，我们的方法无法防止事务执行顺序发生变化。例如，因死锁或系统故障导致事务中止时，事务可能会被重新提交执行。

Strict Serializability. Our approach features three core advantages that provide important foundations for achieving strict serializability. First, since we support one-shot transactions, the system is aware of the keys that will be touched from a transaction prior to its execution. Furthermore, these one-shot transactions can be arranged prior to their execution in a

specific serial order – that order can be set to be the order of arrival, thus guaranteeing strict serializability. Second, Apache Flink, which executes our transactions, recovers from a failure by falling back to the latest completed checkpoint and re-processes input requests following the checkpoint. This strategy allows us to reconstruct the exact same state as prior to the failure under the assumption of deterministic computations. Finally, data-parallel processing in disjoint state partitions allows us to execute concurrent transactions in a parallel manner and without the need for concurrency control.

Relation to Deterministic Databases. Interestingly, the three aforementioned characteristics of our approach resemble design choices opted by deterministic databases [101, 124, 125], which achieve strict serializability: the concurrent processing of a specific set of transactions across a distributed system is guaranteed to result in one, single runtime state.

Furthermore, one could draw inspiration from deterministic databases for advancing its transactional model in two ways. First, transactions on dataflow systems would benefit from an input transaction log for pre-determining the order of transactions in a way that would not introduce aborts during execution, essentially implementing a protocol like Calvin [124]. Second, one could leverage a determinism service [79] to wrap nondeterministic computations, which would cause its state to diverge when recovering from a failure. Essentially, pre-ordering a batch of transactions and ensuring deterministic transaction processing would help dataflow-based transactional FaaS systems guarantee strict serializability.

3.6 Experimental Evaluation

In this section, we describe in detail our experimental evaluation methodology. For the lack of a benchmark aimed at SFaaS, we opted for an extension of the *Yahoo! Cloud Serving Benchmark* (YCSB)[126] benchmark. Furthermore, we go through the experimental evaluation of our system, which is split into six experiments with the following goals.

i) Determine the overhead that function coordination introduced to StateFun (Section 3.6.1).

ii) Compare between the two transaction protocols with/out rollback operations (Section 3.6.2).

iii) Evaluate the system's scalability (Section 3.6.3).

iv) Perform a microbenchmark with a fixed number of machines and a variable number of keys and proportions of *transfer* operations (Section 3.6.4).

v) Compare against the CockroachDB with Kafka clients deployment (Section 3.6.5).

vi) Compare against Beldi (Section 3.6.5).

Regarding resources used, for (i, ii, iv, v), we used three 4-CPU StateFun workers/CockroachDB nodes, and for (iii), each worker had 2 CPUs. In (v), we kept the default settings, meaning that CockroachDB replicates data three times for fault tolerance and high availability. For (vi), we allowed AWS and DynamoDB to autoscale while measuring the maximum concurrency reached by AWS Lambda.

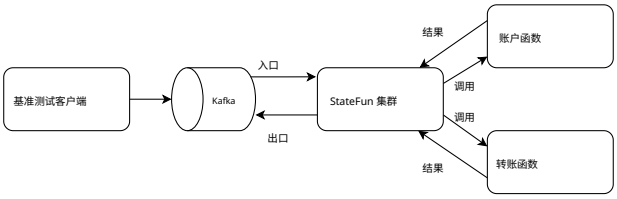


图3.4：StateFun 基准测试设计

基准工作负载

In YCSB, the first step is to insert records into the system with a unique ID and several task-specific fields. After the data insertion stage, the benchmark performs operations on the initialized state. YCSB defines *read* and *write* operations as part of their core workloads. Because this work's main contribution is distributed transactions across stateful function instances, we added a new operation based on an extension introduced in [127]. This operation is called a *transfer*, and it atomically subtracts *balance* from one account and adds this to another, meaning that records also include a numeric *balance* field. These additions mean that the workloads can consist of the following three operations:

read 读取与单个键关联的状态，并将其输出到出口。

write 更新与某个键关联的字段，并向出口输出一条成功消息。

transfer 需要两个键和指定的数量，从一个键的余额中扣除该数量，并将其加到另一个键上。根据事务结果，向出口输出成功或失败消息。

在各项实验中，我们改变每种操作在生成的工作负载中的比例。在 YCSB 中，用户可以选择操作记录标识符的概率分布。在本研究中，我们假设键服从均匀分布。这样做还有一个额外好处：通过减少系统中的记录总数，可以透明地增加针对单个键的请求数量。最后，YCSB 允许改变字段数量以及与每个字段关联的值的大小。在此评估过程中，所有记录都包含十个字段，每个字段含有一个 128 位随机字符串，此外还包含一个整数型字段。我们实现了一个 StateFun 应用程序，其中包含以下两个函数，用于支持第 3.6 节中定义的操作：

– **Account Function.** This is a regular function containing the record state for each key. It processes messages to read the state, updates the fields, and subtracts or adds balance as part of a transaction. It throws an exception and rolls back the transaction if the key does not exist or if there is an insufficient balance to subtract the transaction amount.

– **Transfer Function.** The transfer function is a transactional/coordinator function that takes a message consisting of two different keys and an amount. That message represents a transaction consisting of two function invocations, one to each function key. This function is implemented with both the two-phase commit and the Saga API.

Figure 3.4 depicts the architecture of the system under test. The benchmark publishes the workload to a Kafka cluster. StateFun reads from Kafka as ingress, invokes the

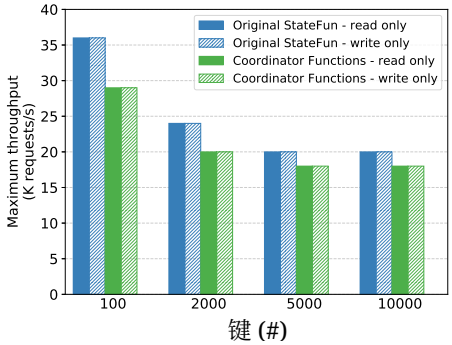


Figure 3.5: Maximum throughput for the original StateFun vs. StateFun with coordinator functions.

appropriate functions, and then publishes the result to a Kafka topic as an egress. For CockroachDB (v21.1.7), Kafka clients read from the relevant topics and submit queries to the non-geo-replicated database.

Although CockroachDB and Kafka can provide exactly-once semantics individually, because the state (CockroachDB) and messaging (Kafka clients) are not managed by a single entity and do not share a single checkpointing mechanism, this deployment offers at-least-once semantics. More specifically, clients that consume Kafka queues that deliver transaction-initiating events need to pull an event from a Kafka topic, submit a query to CockroachDB, and acknowledge the transaction's execution back to Kafka. However, in the event of a client (or database) failure, the transaction may be executed, but the message to the queue may never be acknowledged. Not having returned the acknowledgment to Kafka, the client will re-execute the same transaction after recovery. In general, unless the transactions come with application-specific idempotence keys, the system by itself cannot enforce exactly-once processing guarantees, falling back to at-least-once guarantees.

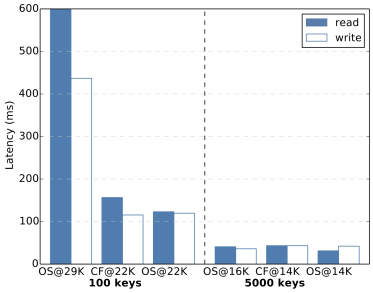
Our StateFun-based implementation and the CockroachDB deployment are deployed on SurfSara[3], an HPC cloud with instances with up to 80-vCPUs. For our experiments, we used a two-VM Kubernetes cluster to simplify the deployment and management of the system's separate components with enough vCPUs to support the system's configuration under test. Beldi was deployed on AWS. All components shown in Figure 3.4 can be horizontally scaled as necessary. Additionally, we give the Kafka cluster and the clients enough resources to ensure that they can handle the load: when a bottleneck appears, it can be attributed to the system performing the application logic, i.e., the StateFun cluster, CockroachDB, or Beldi's API.

评估指标

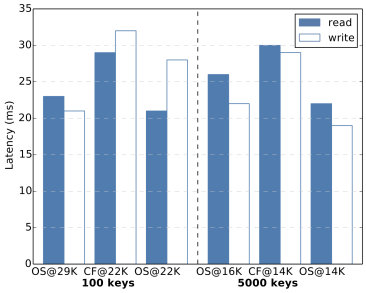
We evaluate the systems based on two metrics. First, the throughput is either at max or stable (80%), showing the number of workload operations the system can handle per second, and the latency, showing the time it takes to process an operation.

The maximum throughput of each workload and system configuration is found by

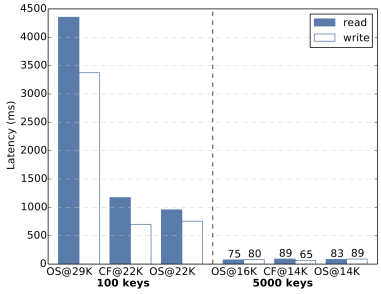
[3] https://userinfo.surfsara.nl/systems/hpc-cloud



(a) 均值



(b) 中位数



(c) 95th percentile

Figure 3.6: Graphs comparing latencies of original StateFun (OS) and StateFun with coordinator functions (CF) at different throughputs for read-only and write-only workloads.

steadily increasing the input throughput created by the benchmark clients in Kafka until the StateFun cluster/CockroachDB can no longer consistently handle the load, as measured by the system's output throughput in Kafka. At some point, the output throughput starts fluctuating, and we define this value as the maximum throughput for the configuration. In the comparison with Beldi, we could not measure it this way since it will always rescale to accommodate the new load. So our approach in this matter is to take the 80% throughput of the StateFun configuration and run Beldi with the same input throughput.

We use the Kafka event time for the ingress and egress events of operations to measure their end-to-end latency. Since latency is always dependent on the throughput, in our experiments, we set the throughput to 80% of the maximum throughput to allow consistent operation of the system under test and measure the latency accurately. When comparing latencies, the different throughput rates at which the latency is measured should be considered.

3.6.1 Coordination Overhead

In the first experiment, the performance of StateFun with coordinator functions is compared against the original on non-transactional workloads to see how much computational overhead the coordination logic has added. In Figure 3.5, we show the maximum throughput achieved by the two systems for a varying number of keys. While in Figure 3.6, we show the

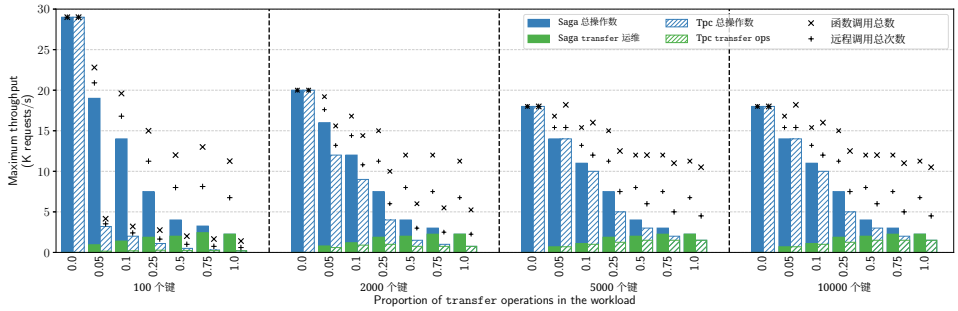


图 3.7：工作负载中传输操作比例不断增加时的最大吞吐量

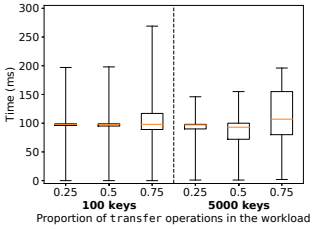
different latencies for the systems across *read* and *write* workloads at different throughputs and numbers of keys.

吞吐量。首先可以观察到，在 100 个键的情况下，吞吐量下降了 20%；随着键的数量增加，降幅趋于稳定，约为 10%。性能下降是因为，与原有的仅追加方法相比，批处理机制更为复杂，它通过两阶段提交事务来确保函数调用彼此隔离。此外，协调器函数会跟踪事务进度，这也会带来一些开销。另一个观察结果是，只有读取操作或只有写入操作的工作负载之间，吞吐量没有明显差异。这是因为在 StateFun 中，这两种操作都需要访问远程函数，因此通信层成为瓶颈。

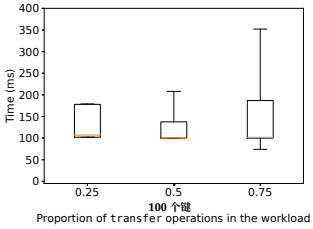
延迟。图 3.6 显示，对于读取工作负载，我们的 StateFun 版本的延迟约高出 20%。然而，随着键的数量增加，差异逐渐缩小至 7%。性能下降是由函数协调所需的额外逻辑造成的。另一个有趣的观察结果是，写入工作负载的性能没有变化。原因在于，StateFun 会在序列化之前对每个读取操作进行批处理，批次越大，累积的开销也越多。相比之下，对于写入操作，只需序列化最后一个版本。此外，两种操作的序列化都发生在远程函数中，这解释了为何它不会影响吞吐量，却会影响延迟。最后，我们认为，为系统引入功能完备的事务执行原语所付出的开销是合理且较低的代价。

3.6.2 Sagas vs Two-Phase Commit

The second experiment shows a performance comparison between the two implemented transaction protocols, their impact on the maximum throughput in perfect conditions (Figure 3.7), and with failures, measuring the impact of locking for the two-phase commit (Figure 3.8) and of rollbacks (Figure 3.9) for the Saga protocols. In these experiments, we set a certain proportion of the workload to be *transfer* operations, and the remaining proportion is equally shared between *read* and *write* operations. In our case, each *transfer* operation causes three remote function invocations (coordinator function and one function per account holder taking part in the transfer). When evaluating two-phase commit functions, we do not include messages sent to detect deadlocks in the total number of invocations.



(a) Time regular locks are held



(b) Time to detect deadlocks

Keys	传输比例	死锁 / 传输操作
100	0.25	9/12014 (0.07%)
	0.5	27/24107 (0.11%)
	0.75	82/35875 (0.22%)
5000	0.25	0/60121
	0.5	0/120089
	0.75	0/179794

(c) Frequency of deadlocks

Figure 3.8: Details of locking behavior for a workload for 100 and 5000 keys with various proportions of transfers without rollbacks. The boxplots show the 5th and 95th percentiles.

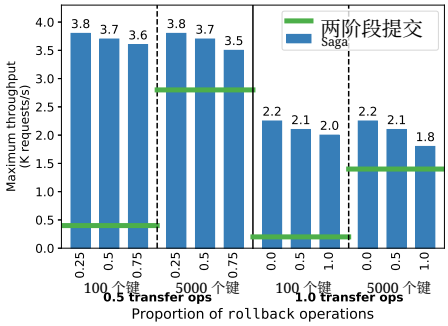


Figure 3.9: Throughput with different proportions of rolled back *transfer* operations for workloads with 50% and 100% *transfer* operations

Therefore, the indicator should be considered a lower bound on the actual number of messages. Finally, we used a uniform key access distribution for these experiments. At the same time, in some real-world scenarios, this can be skewed (e.g., lots of transactions on very active accounts vs. a long tail of inactive ones).

图 3.7 展示了基准测试为账户提供充足余额以确保所有事务均成功时，工作负载中 转账 操作的绝对数量变化时所达到的吞吐量。图中还标示了内部函数调用总量的绝对数量（其中考虑了事务所需的额外内部调用），以及远程函数调用总量的绝对数量。我们观察到，当键数量较少（100 和 2000）时，Saga 的表现远优于两阶段提交。这有两个原因：i）Saga 不需要隔离，因此仍能受益于 StateFun 的批处理机制；ii）两阶段提交中的锁定严重限制了吞吐量。不过，有趣的是，当键数量较多（5000–10000）时，两阶段提交的表现与 Saga 相当，尽管它提供了强得多的保证。这是因为单个函数上的争用较少，从而减弱了锁定的影响，而批处理则毫无益处，如图 3.5. 图 3.7 还显示了另一个现象：随着事务比例上升，函数调用总数仍会下降。这是因为

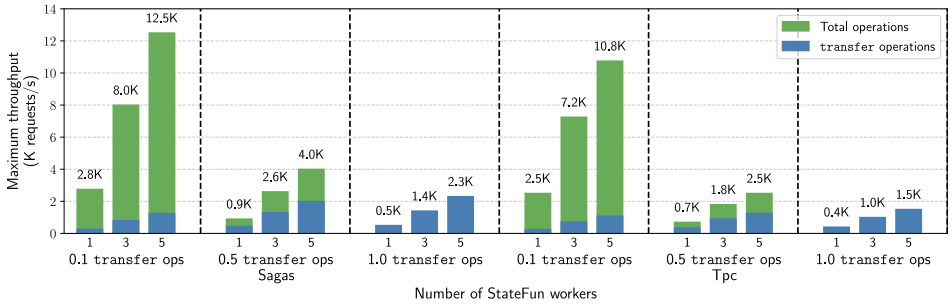
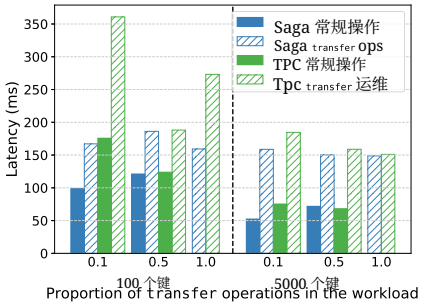


图 3.10：对于不同操作比例的转账工作负载，系统在 5000 个键时针对不同数量的 StateFun 工作进程所达到的最大吞吐量

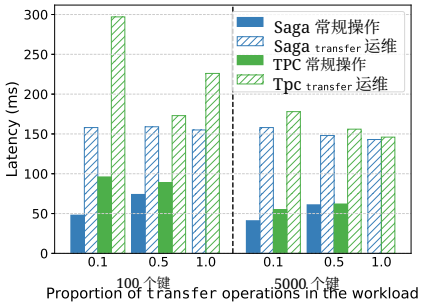
函数调用总数包含了协调事务所需的额外消息传递，因此，转账操作比例较高的工作负载整体吞吐量相对较低。

锁定开销。在图 3.8中，我们测量了锁定行为以及伴随两阶段提交协议出现的死锁。锁定持续时间是从函数实例向 准备消息发送响应的时刻，到收到提交或中止消息并向远程函数发送下一批次的时刻为止。在图 3.8a中，我们看到不同工作负载的中位数几乎没有差异；但当转账操作的比例较高时，较高百分位数会显著增加。接下来，我们测量死锁频率。图 3.8c 展示了工作负载中死锁数量与转账操作总数之间的关系。正如预期，使用 5000 个键的工作负载不会出现死锁，因为争用较低。对于 100 个键，随着转账操作比例增加，我们观察到死锁数量也随之增加。不过，死锁占所有转账操作的比例仍然很小。最后，图 3.8b 展示了检测死锁所需的时间，即执行 Chandy-Misra-Haas 算法所需的时间。我们观察到，所有工作负载下的检测时间中位数都相近；同时也可以看到，随着转账操作数量增加，较高百分位数对应的时间也会增加。

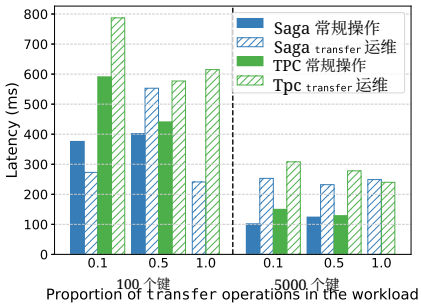
回滚开销。图3.9 展示了在 50% 和 100% 传输操作负载下，当不同传输操作比例发生失败时，Saga 和两阶段提交协调器函数的最大吞吐量。正如预期，使用两阶段提交时，回滚不会增加系统负载，因为无论如何，协调器函数都需要发送第二条消息。随着需要回滚的传输操作比例上升，也没有出现异常情况。由于协议要求在系统中发送额外的补偿消息，吞吐量有所下降。不过，在使用 5000 个键时，50% 传输操作下的差异很小：回滚比例从 25% 增至 75% 时，吞吐量仅下降 8%；在 100% 传输操作下，降幅增至 18%。这比使用 100 个键时的降幅更大；后者仍可利用 StateFun 的批处理机制，将最差情况下的性能下降限制在 10%。尽管 Saga 协议的补偿操作会导致性能下降，但即使在 5000 个键且回滚比例为 75% 的最差情况下，其速度仍比两阶段提交快 20%。



(a) 均值



(b) 中位数



(c) 95th percentile

100 个键						5000 个键					
Saga			Tpc			Saga			Tpc		
0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0
11K	3K	2K	1.5K	0.38K	0.16K	9K	3K	2K	8K	2K	1.2K

(d) Throughputs at which latency was measured

Figure 3.11: Graph comparing latencies for Sagas and two-phase commit coordinator function for different keys and transaction proportions in the workload at 80% of the respective maximum throughputs

3.6.3 Scalability Comparison

在最后一项实验中，我们评估了所提出的、采用协调器函数的系统的可扩展性。在图 3.10 中，我们展示了两阶段提交和 Saga 在不同数量的 StateFun 工作进程以及工作负载中不同事务比例下的最大吞吐量。对于 Saga，在所有工作负载下，工作进程从 1 个增至 5 个时，可扩展性均接近 90%。对于两阶段提交，当传输操作占比为 10% 时，工作进程从 1 个增至 5 个的可扩展性为 87%，当占比达到 100%传输操作时则降至 75%。

The reason for the low decrease in scalability on both protocols is that as the number of workers increases, more traffic needs to go over the network. In the Sagas' case, the efficiency does not decrease across all workloads for the same reasons as expressed in Section 3.6.2. Namely, the system can still utilize batching, no locking is required, and the number of messages is two times lower than the two-phase commit protocol when all transactions succeed. On the other hand, the 8% decrease in scalability in two-phase

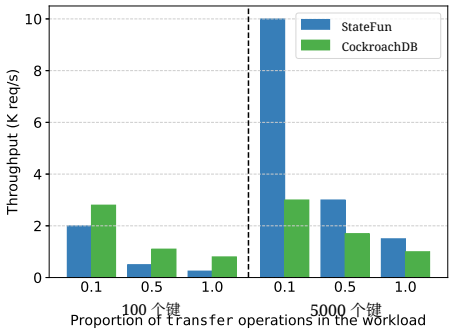


图 3.12：比较 CockroachDB 和 Flink StateFun 在不同传输 操作比例下的最大吞吐量（其余操作以相同概率为读取 或更新 操作）。两个系统均部署了 3 个实例，每个实例配备 4 个 CPU。

commit from 10% to 100% *transfer* operations is due to the protocol's requirements for locks, more messages, and the inability to use batching. Considering all the impending factors, it still achieves decent efficiency with strong consistency guarantees in fully transactional workloads.

3.6.4 微基准测试

作为最后一项实验，我们对系统进行了微基准测试。首先，我们固定资源数量，然后针对每种传输 比例和键数量，在 80% 负载下测量吞吐量及相应的延迟。从图 3.11 所示结果可以看出，对于键数量较少的使用场景，Saga 在吞吐量和延迟方面都大幅胜过两阶段提交协议：性能提升超过 650%，延迟至少降低一半。键数量增加后，争用的影响会减弱。我们观察到两种协议之间的差距有所缩小：吞吐量平均相差约 40%，延迟差距则稳定在 20% 左右。总而言之，如果业务逻辑允许，那么对于键数量较少或争用较高的场景，Saga 显然是更好的选择。其他情况下，选择主要取决于一致性保证要求，因为两者的差异并不显著。

3.6.5 Comparison Against the State-of-the-Art

CockroachDB. We compare the performance of StateFun against a production-grade distributed database, CockroachDB, in terms of throughput and latency. Due to the fundamental differences between the two systems, this is merely a reference comparison. In this experimental setting, the input requests consist of a varying proportion of transactional and non-transactional requests. We signify transactional requests as transfer operations and non-transactional requests as non-transfer operations.

As Figure 3.12 shows, CockroachDB outperforms StateFun in terms of throughput by a constant factor when transactions are evoked on a small number of unique keys. In addition, this experiment configuration examines the performance of the two systems

100 个键				
操作	StateFun		CockroachDB	
	中位数	第95百分位	中位数	第95百分位
Transfer (0.1)	297	787	48	86
非传输	96	591	47	79
Transfer (0.5)	173	577	22	72
非传输	89	441	17	68
Transfer (1.0)	226	615	41	114
5000 个键				
操作	StateFun		CockroachDB	
	中位数	第95百分位	中位数	第95百分位
Transfer (0.1)	178	308	36	113
非传输	55	150	21	107
Transfer (0.5)	156	278	21	64
非传输	62	129	11	62
Transfer (1.0)	146	240	43	78

Table 3.2: Latency compared for StateFun and CockroachDB, each system was run at 80% of the maximum throughput measured as shown in figure 3.12

when there is high lock contention since subsequent transactions on the same key have to wait for previous ones to complete. Notably, the performance difference in terms of throughput remains the same while the proportion of transactions in the input request set increases from 0.1 to 0.5 to 1. When there are many keys (e.g., 5000), StateFun outperforms CockroachDB. In fact, for a small proportion of transactions (0.1), StateFun achieves four times more throughput. As the number of transactions grows, the performance difference shrinks. These results can be explained by a more sophisticated or aggressive batching mechanism that enables StateFun to efficiently batch non-transactional requests. When there are many non-transactional requests, the effect of batching provides a significant performance advantage, which is shrinking as the number of non-transactional requests becomes smaller.

On the other hand, CockroachDB is superior in terms of latency performance as Table 3.2 depicts. Both median latency and latency at the 95th percentile are roughly six times better on average than StateFun's in all configurations. This result can be explained because CockroachDB is run with default settings, and there is no batching implemented at the application level. This contributes to lower throughput, but it also favors lower latency. On the other hand, StateFun can inherently apply batching at several points in the system, such as when i) it sends a request to a remote function, ii) fetches requests from Kafka, and iii) produces responses to Kafka.

In summary, CockroachDB seems more suitable for handling skewed transactional workloads, although the performance improvement against StateFun is constant in terms of throughput. Thus, a potential superiority based on the locking mechanism of CockroachDB is capped and does not result in a scalable advantage. Furthermore, CockroachDB replicates data three times, leading to additional overhead but providing the capacity to serve requests even in the case of node failures. On the other hand, StateFun provides no replication and needs to recover from a checkpoint following a node failure. On the other hand, StateFun can leverage its sophisticated batching mechanism to drive significantly better throughput for workloads containing a modest number of transactions. Notably, while CockroachDB supports full transactional SQL and StateFun supports only one-shot functions, due to the simplicity of the workload, the feature set should not have a significant impact on

100 个键							
操作	**吞吐量**	**StateFun**			**Beldi**		
		CPU	中位数	第95百分位	最大并发数	中位数	第95百分位
Transfer (0.1)	1.5K	80	298	723	128	49	739
非传输			99	572		83	693
Transfer (1.0)	0.16K	80	223	532	182	91	724
5000 个键							
操作	**吞吐量**	**StateFun**			**Beldi**		
		CPU	中位数	第95百分位	最大并发数	中位数	第95百分位
Transfer (0.1)	8K	80	184	287	1000*	123	174
非传输			52	184		50	77
Transfer (1.0)	1.2K	80	146	273	902	114	847

Table 3.3: Comparison between latencies of Beldi and StateFun (*experiment is throttled and runs at a lower throughput ≈ 4K, i.e., experiment lasted longer)

performance. In addition, the executed workloads allow for less locking and more batching. Finally, CockroachDB demonstrates reliably low latency in all configurations, roughly six times lower than StateFun.

Beldi。我们还将 StateFun 与有状态函数即服务库和运行时 Beldi 进行比较。Beldi 运行在 AWS Lambda 上，并使用 DynamoDB 作为事务的后端存储。由于无服务器环境错综复杂，且可配置方式受限，我们将比较范围限制为固定吞吐量请求数量下的延迟性能，因为我们无法充分了解 Beldi 使用的资源数量。AWS 仅公开 Lambda 函数的并发级别，并允许将其限制为某个最大值。在图 3.3中，最大并发量指 AWS Lambda 所使用的最大并发量。在每次实验过程中，最大并发量都相当稳定。值得注意的是，没有关于所用底层硬件规格的信息。

此外，即使是延迟性能也无法提供公平的比较，因为 Beldi 只测量从请求开始执行到执行完成所花费的时间，而不考虑请求开始执行前用于路由和在输入队列中等待的时间。因此，Beldi 因负载过高而出现的性能限制不会反映在测得的延迟中。我们尝试通过测量实验完成时间来弥补这一点，并估算 Beldi 的实际吞吐量。另一方面，我们在 IaaS 云基础设施上运行 StateFun，为其提供特定数量的计算资源，并测量端到端延迟。两种基础设施和实验设置之间的差异，限制了我们能够从中获得的洞见。

图3.3 展示了实验结果，我们从中得出了关于延迟的两个值得注意的观察结果。对于非转账操作，无论键的数量是多少，StateFun 和 Beldi 都能实现同样低延迟的性能。对于转账操作，Beldi 的中位延迟性能优于 StateFun 2 至 3 倍；而第95百分位数的尾延迟则表明，两个系统之间没有显著差异。在 Beldi 中，延迟仅涵盖系统内部产生的延迟，这些延迟可能源于 Beldi 内部的锁竞争、Lambda 函数与 DynamoDB 之间的通信停滞，以及 DynamoDB 中的排队。遗憾的是，我们无法确定观测到的尾延迟究竟由哪些因素导致，也无法量化各因素的影响。

最后，实验中的一个重要因素是，Beldi 可以自由地自动扩容，最多支持 1000 个并发执行的函数。这种大幅增加资源可用性的做法，所提供的资源远超分配给在 StateFun 上执行的远程函数的 80 个 CPU。值得注意的是，当唯一键的数量较多、锁竞争较低时，这种并发度不足以满足每秒 8K 个请求的输入吞吐量。在这种情况下，AWS Lambda 用尽了所有可用的并发容量，请求执行受到限流，只能等待 CPU 可用。考虑到实验持续时间，我们估算 Beldi 达到的吞吐量约为每秒 4K 个请求。请注意，Beldi 的延迟不受影响，因为它不计入排队等外部延迟。另一方面，我们观察到，当唯一键的数量较少、锁竞争较高时，Beldi 的效率相当高。它使用的并发度高于 StateFun 可用的并发度，但总体数量级相同。Beldi 的高效可能得益于它能在可立即执行的请求与需要休眠、直到获得继续执行所需锁的请求之间灵活调度。

最后，Beldi 的实测性能未计入垃圾回收的影响。Beldi 配备了垃圾回收器，可定期缩减其事务日志；但由于本文所述实验的持续时间过短，实验期间无需运行垃圾回收器。一般而言，垃圾回收器预计会带来额外开销，而这部分开销未体现在我们的一系列实验中。

3.7 相关工作

SFaaS 系统。 SFaaS 一直是研究界和开源社区都非常活跃的领域。在研究界，最相关的工作是 Beldi [13], 它与 AFT 一样[128], 基于亚马逊的 AWS Lambda 构建，增加了容错和事务支持，从而能够实现更复杂的状态管理。两者的主要区别在于，Beldi 的执行环境完全是无服务器的，而 AFT 依赖外部服务器提供事务支持。为此，Beldi 使用原子日志记录，并扩展了 Olive [129], 以确保读写操作具备容错能力，同时通过垃圾回收来管理日志增长。对于事务，Beldi 支持一种两阶段提交协议的变体，通过等待-死亡死锁预防来强制实现强一致性保证。Cloudburst 与 Hydrocache [114] 在同一 DAG 工作流中提供因果一致性保证，其后端由 Anna [115], 键值状态后端提供。另一个很有前景的 SFaaS 系统 FAASM[130], 支持函数之间的直接内存访问，同时保持隔离；与容器相比，它还能缩短初始化时间。本文撰写时，FAASM 尚不支持事务。最后，两个最知名的开源 SFaaS 项目是 Cloudstate[4]，（基于有状态 Actor）和 Apache Flink StateFun，后者将在第 3.2 节详细介绍。在 Cloudstate 中，同一集群内不同 Actor 之间以及用户定义函数之间可通过 gRPC 进行通信，并保证至少一次处理。

Transactional Programming Model. The most notable difference among these systems in terms of programming model is state access. Both StateFun and Cloudstate encapsulate state within a specific function instance. In contrast, Cloudburst and Beldi allow

[4] https://cloudstate.io/

any function access to any state stored in Anna or DynamoDB, respectively. Regarding transactions, only Beldi offers a programming model where the developer writes two markers (begin/end_tx), and every function invocation in between will execute as part of a transaction. Our contribution is a programming model that supports transactions on StateFun with the choice of strong or relaxed consistency guarantees.

流处理事务。 此外，文献中也对基于流处理系统的事务给予了一定关注。在 [131], 中，作者提出了一个同时适用于数据流和传统表格数据的事务模型。沿用类似模型，在 [132], 中，作者为分区状态之间的快照隔离和一致性增加了保证。随后，TSpoon [133]，（FlowDB [119] 的扩展）提出了一种构建于流处理器之上的数据管理系统，该系统支持事务，并提供强事务保证和弱事务保证以及可查询状态两种选择。我们的工作聚焦于在无服务器数据流系统上执行的通用有状态函数之间的事务工作流。

分布式数据库。 上述流处理系统与分布式数据库有着相同的主要目标 [49, 50, 121, 134–136], , 即如何扩展到多台机器，同时提供可串行化事务保证。这是数据库研究中的一个老问题。R* 系统 [134] 是最早尝试采用带分布式死锁检测的两阶段提交协议的系统之一。随后，H-Store [121] 等较新的方法表明，当事务仅涉及单个分区时，分布式数据库解决方案可以同时提供极高的性能和事务保证。目前，分布式数据库研究围绕全球分布式数据库展开，Spanner [49] 基于原子钟，通过跨所有位置/机器的时间戳机制引入了可串行化事务。此外，Carousel [135]和 SLOG [136] 等方法改进了全球分布式数据库事务。Carousel 通过尽量减少网络使用量来增强事务执行，而 SLOG 则提供了一种基于数据与客户端之间邻近性的细粒度事务协议。最后，CockroachDB [50] 无需复杂的时间机制即可提供可串行化的全球分布式事务。

Benchmarks. The large variety of use cases and systems makes them difficult to compare using a standardized benchmark. The related benchmarks that could be used to evaluate SFaaS systems are the *Yahoo! Cloud Serving Benchmark* (YCSB)[126] and the DeathStarBench [137]. Given that StateFun is based on Flink, which is a stream processing system, a stream processing benchmark [138] would be another alternative. However, its workloads are not representative of those executed by an SFaaS system. In addition, we did not consider TPC-C [139] because it was created to test relational database management systems, including transactions, and requires many additional features not present in SFaaS. We ultimately chose to develop and use an extension of YCSB [127] that introduced explainable transactional workloads, allowing for an easier interpretation of the results.

3.8 Discussion & Open Problems

Programming Models for the Cloud. Although the stateful dataflow model has been very successful as an execution model, it has not been leveraged thus far as an intermediate representation. Historically, MapReduce/Hadoop [140] and Dryad [30] were first proposed

as a means of authoring and executing distributed data-parallel applications using high-level language constructs, such as Java functions and LINQ [141] respectively. Many systems have followed that execution model subsequently, including streaming dataflow systems such as Apache Storm [142], Flink [8], Naiad [143]. However, none of these systems could execute general-purpose cloud applications; their programming model focuses on distributed collection processing and adopts a functional programming API.‘

云应用的数据流。我们认为，有状态函数等抽象可以在数据流引擎中充当高级编程模型，并对云编程产生重大影响。目前，云编程的方法要么是使用 Bloom 等领域特定语言（DSL），[29], Hilda[31], 和 Erlang [104], ，要么是将其作为 Akka [105], Spring Boot（spring.io）等主流语言中的库。这里的主要观察是，开发者要么必须学习一种新的领域特定语言，要么使用会将实施细节泄漏到业务逻辑中的库。虚拟参与者与 Orleans [27, 36] 与本工作的理念非常接近，Apache Flink 的 StateFun 也从中汲取了灵感。然而，Orleans 需要专用运行时，并且不提供精确一次函数执行。正如本文所展示的，由于我们受益于现代数据流系统的状态管理和精确一次保证，因此实现非常复杂的协议（包含大量边界情况）可能会更加简单。如今，数据流系统已得到充分理解，具备可扩展性和一致性；因此，我们相信它们将在未来的云执行引擎中发挥关键作用。

Future Dataflow Systems. However promising they can be, dataflow engines still suffer from several issues. Stream processors such as Apache Flink [8], or Jet [113] have been designed for continuous operation on high-throughput streams. However, stateful functions have very different workload characteristics. For instance, lots of cloud applications may have to call external services – a source of non-determinism [79], and functions calling other functions, expecting return values, introduce cycles in the dataflow graph. Current dataflow systems either do not support cycles or support a few special cases of cycles. This is because cycles can cause deadlocks and various other issues [78, 144] that need to be dealt with. Finally, in this paper, we introduced transactions at the function level without having to touch the core of Apache Flink's dataflow engine. However, proper implementation of transactions would require the dataflow system itself to be aware of transaction boundaries (e.g., commit, prepare) and incorporate transaction processing into its fault-tolerance protocol. We think that more research needs to be performed to get dataflow systems fully capable of leveraging their potential.

3.9 结论

本章探讨了如何在无服务器平台上支持跨云应用的事务工作流。这一问题在微服务和云应用领域中广为人知。此外，我们还提出了一种编程模型及其相应的实施方案，用于编写跨有状态无服务器函数的工作流，并提供可配置的事务保证。开发者可以选择在函数之间执行具有严格原子性和一致性保证的分布式事务，也可以选择提供最终原子性和一致性的 Saga 工作流。这些互补的方案能够真实地反映现实世界用例的需求。我们介绍了基于

Apache Flink StateFun 的实施方案，并在我们开发的扩展版 YCSB 基准测试上对其进行了评估，评估内容包括：a) 相较于原始 StateFun 的吞吐量和延迟开销；b) 分布式事务与 Saga 工作流之间的性能效率；以及 c) 可扩展性。我们发现，事务工作流为系统增加的开销约为 10%，仍在可接受范围内；根据系统中正在进行的事务工作流数量，Saga 的性能显著优于分布式事务，优势为 15%–34%；在可扩展性方面，Saga 达到了 90%，而两阶段提交为 75%–87%。此外，与无服务器 SFaaS 运行时的比较表明，我们的方案能够实现更高的吞吐量，但也会带来更高的延迟。最后，我们将其与流行的分布式数据库 CockroachDB 进行了比较，后者在高争用场景下以及延迟方面表现更佳。值得注意的是，在键分布稀疏的情况下，我们的方案取得了更好的结果；同时，与我们部署的以 CockroachDB 为后端、采用至少一次语义的 Kafka 相比，我们的方案提供了精确一次处理语义。

4

Stateflow: a Domain-Specific Language for General-Purpose Cloud Applications

Chapter 3 examined how transactional guarantees can be integrated into existing stateful function-as-a-service platforms, focusing on extending Apache Flink Statefun. While this approach demonstrated the feasibility of augmenting runtime environments with strong consistency guarantees, it did not address the equally important challenge of programmability. Cloud developers continue to face significant complexity when translating high-level application logic into distributed execution semantics. In this chapter, we turn our attention to this challenge.

We introduce Stateflow*, a domain-specific language and compiler pipeline that enables developers to author general-purpose cloud applications using familiar object-oriented constructs. Stateflow compiles such programs into a dataflow intermediate representation that is portable across multiple execution backends. This chapter details the design of the programming model, its compilation strategy, and the execution guarantees it preserves, thereby advancing the thesis's objective of democratizing cloud applications.*

This chapter is based on the following work:
K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows, EDBT'24 (vision) and CIDR'23 (abstract) [55].

Organizations nowadays enjoy reduced costs and higher reliability, but cloud developers still struggle to manage infrastructure abstractions that leak through in the application layer. As a result, managing application components, such as service invocation, messaging, and state management, requires much more effort than developing the application's business logic [47]. Worse, moving a cloud application between cloud providers is prohibitive due to significant differences in the underlying systems.

尽管分布式应用程序编程存在多种方法（例如 Bloom [29], Hilda [31], Cloudburst [11], AWS Lambda、Azure Durable Functions 和 Orleans [27, 36]），但在实践中，开发人员主要使用通用编程语言的流行库，例如 Java 中的 Spring Boot 和 Python 中的 Flask。

这些方法都无法提供处理保证，因而不支持 精确一次处理：即系统能够恰好一次地反映消息对状态所产生的更改。相反，它们提供至多一次或至少一次处理语义。因此，程序员不得不在业务逻辑中“污染”一致性检查、状态回滚、超时、重试和幂等性[2, 109]。

我们认为，无论采用何种云编程方式，除非执行引擎能够提供精确一次处理保证，使其能够在编程模型层面得到假定，否则我们永远无法将分布式系统相关问题的负担从程序员身上移除。据我们所知，截至撰写本文时，唯一能够保证精确一次消息处理 [78, 79] 的系统是批次 [30, 140, 145]和流式处理 [8, 142, 143]数据流系统。然而，它们的编程模型遵循函数式数据流 API 范式，使用起来很繁琐，需要进行培训，并对开发人员倾向于用来表达应用逻辑的典型命令式代码进行大量重写。

For these reasons, we argue that the dataflow model should be used as a low-level intermediate representation (IR) for the modeling and executing distributed applications, but not as a programmer-facing model. Technically, one of the main challenges in adopting a dataflow-based IR is that the dataflow model is functional, with immutable values propagating across operators that typically do not share a global state. Hence, adopting a dataflow-based IR entails translating (arbitrary) imperative code into a functional style. Compiler research has systematically explored only the opposite direction: to compile code in functional programming languages into a representation that is executable on imperative architectures – like modern microprocessors. Yet, the translation from imperative to functional or dataflow programming remains largely unexplored.

This chapter presents a prototypical programming model and an IR that compiles imperative, transactional object-oriented applications into distributed dataflow graphs and executes them on existing dataflow systems. Instead of designing an external Domain-Specific Language (DSL) for our needs, we opted for an internal DSL embedded in Python - a popular language for cloud programming. Specifically, a given Python program is first compiled into an IR, an enriched stateful dataflow graph independent of the target execution engine. That dataflow graph can then be compiled and deployed to various distributed systems. The current set of supported systems includes Apache Flink Statefun [25] and Styx (Chapter 5). The choice of a runtime system is entirely independent of the application layer, which allows switching to different runtime systems with no changes to the application code.

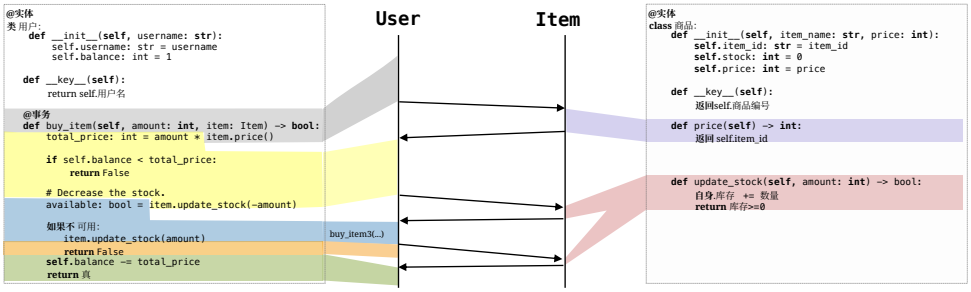


Figure 4.1: Two stateful entities: *User* and *Item*. The content of imperative functions is split into multiple functions that access the common state of a given entity. Those functions are then encoded into a stateful dataflow that can be executed in a distributed streaming dataflow engine. As a result, *i*) imperative code is executed in an event-based manner without the need to block, and *ii*) the code retains exactly-once processing guarantees without the need for programmers to write failure-handling code such as state management, call retries or idempotency.

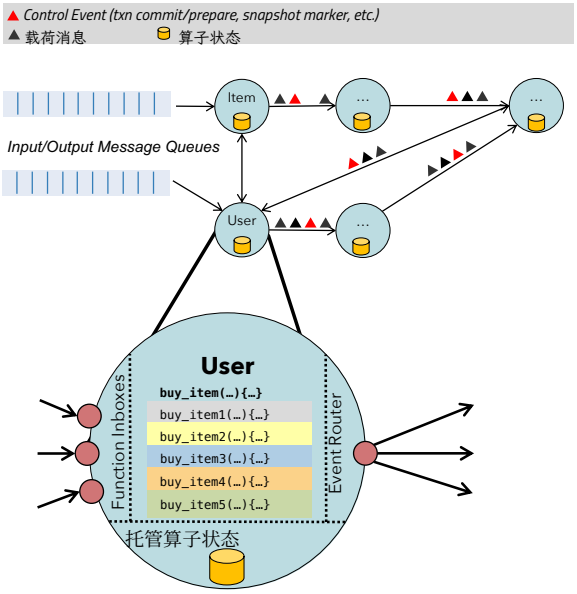
The contributions of this chapter go as follows:

- To the best of our knowledge, this is the first work to propose compiling and executing imperative programs into distributed, stateful streaming dataflows.
- We present a compiler pipeline that analyzes an object-oriented application and transforms it into an IR tailored to stateful dataflow systems.
- We describe an IR for cloud applications and how that IR translates to a dataflow execution graph, targeting various distributed systems, thereby making cloud applications portable across different systems and infrastructures.
- We compare Stateflow, a novel transactional dataflow system, against Apache Flink Statefun and demonstrate the limitations of existing dataflow systems, motivating further research. Our experimental evaluation shows that Stateflow incurs low latency in the YCSB+T [127] workload.

The proposed programming model presented in this chapter can be found at: `https://github.com/delftdata/stateflow`.

4.1 From Imperative Code to Dataflows

Historically, imperative programming and functional programming have evolved in parallel: imperative as a direct codification of (operational) computational models (e.g., Von Neumann architecture, Turing machines) and functional inspired by mathematical abstractions (e.g., lambda calculus, program denotation). While functional programming has been embraced by several languages (e.g., Haskell [146], ML [147]), imperative programming has taken the scene, with most mainstream languages featuring object-oriented (mutable) abstractions. Over the last few years, imperative languages like Java and Python, which support various domain-specific packages, e.g., networking, statistics, numeric computation, etc., have become extremely popular among non-expert programmers.



F图 4.2：由五个实体构成的逻辑数据流图，重点展示了图 用户 实体，该实体见图 4.1。

Yet, the benefits of functional programming have been known for a while. Most notably, functional code is often *embarrassingly parallelizable* because of the lack of side effects and mutability. Developers working with imperative languages – let alone non-expert developers – can hardly access this feature.

4.1.1 方法概述

我们的编译器流水线的核心原则是，开发者只需为 Python 类添加 *@stateflow* 注解，系统便会自动分析并转换这些类，生成中间表示，再将其转换为有状态数据流图，以便部署到数据流系统上。与（虚拟）Actor 类似，[36, 105], 实体可以调用其他实体的方法。图4.1 展示了两个示例实体： 用户和 商品。编程模型的详细信息见第 4.1.2 节。

在第一轮抽象语法树（AST）静态分析中，我们提取类的变量（即通过 *self* 引用的实例属性）、各个方法的名称，以及程序员指定的所有相应类型（第 4.1.2 节）。在第二轮分析中，我们识别彼此交互的类，以构建函数调用图（第 4.1.3 节）。随后分析调用图，以识别对其他函数（可能位于远程机器上）的调用；此时必须拆分函数，从而构成最终的数据流（第 4.1.4 节）。

This dataflow graph, enriched with the compiled classes, execution plans, and all metadata obtained from static analysis, comprises the intermediate representation (Section 4.1.5). Finally, that intermediate representation can be translated, deployed, and executed in different target systems (Section 4.2).

4.1.2 Programming Model & Limitations

Expressiveness. Our programming model allows programmers to specify simple, object-oriented Python programs. Classes can have references to other classes and call their functions. We term an instance of such a class as a *stateful entity*. The Stateflow compiler currently can analyze conditionals, *for*-loops that iterate through Python lists, as well as general *while* loops.

Limitations. Stateflow requires static type hints for the input/output of stateful entity functions and ensures the existence of those hints via a static pass over the analyzed classes. Moreover, the functions cannot be recursive. Another assumption that Stateflow makes is that each entity contains a *key()* function. This *key()* function is used by a routing and translation mechanism to partition and distribute the load among parallel instances of that entity within a cluster. Furthermore, the key of a stateful entity cannot change throughout that entity's lifetime. Finally, the entities' state needs to be serializable, i.e., connections to databases, local pipes, and other non-serializable constructs are not allowed and will eventually generate a runtime error.

Running Example. Figure 4.1 contains the code for a User and an *Item* entities. Note that since *Item* is a stateful entity, a call to *item.update_stock(...)* is a remote function call. Both *User* and the *Item* entities are partitioned across the cluster nodes, using the given entity's *key* function.

4.1.3 From Entities to Dataflow Operators

Each Python class translates to an operator (also called a vertex) in the dataflow graph. In a dataflow graph, an operator cannot be "called" directly, like a function of an object. Instead, an *event* has to enter the dataflow and reach the operator holding the *code* of that entity (e.g., the *User* class) as well as the actual *state* of the entities that instantiate the class (e.g., the *balance* and *username* of the *User* in Figure 4.1).

Specifically, each dataflow operator can execute all functions of a given entity, and the triggering function depends on the incoming event. Since operators can be partitioned across multiple cluster nodes, each partition stores a set of stateful entities indexed by their unique key. When an entity's function is invoked, the entity's state is retrieved from the local operator state. Then, the function is executed using the arguments found in the incoming event that triggered the call, as well as the state of the entity at the moment that the function is called.

Example. A *User* operator as seen in Figure 4.2, is partitioned on *username*. Upon invocation of a function of the *User* entity, an event is sent to the dataflow graph's input queues. The incoming event is partitioned on *username* by an ingress router. Via the dataflow graph, the event ends up at the operator storing the state for that specific *User*. The system then reconstructs the *User* object using the operator's code and the function's state and executes the function. Finally, the function return value is encoded in an outgoing event forwarded to the egress router. This egress router determines if the event can be sent back to the client (caller outside the system, such as an HTTP endpoint) or if it needs to loop back into the dataflow to call another function.

The Need for Function Splitting. For simple functions that do not call other remote functions, both the translation to dataflows and the execution are straightforward. However,

if the function *User.buy_item* calls the (remote) function *item.update_stock* whose state lies on a different partition, the situation becomes more complicated. Note that a streaming dataflow should never stop and wait for a remote function to complete and return before moving on with processing the next event. Instead, it must "suspend" the execution of, e.g., *buy_item* of Figure 4.1, right at the spot that the remote function *item.price()* is called until the remote function is executed. An event comes back to the *User* operator with a return value.

为此，我们采用一种技术，将命令式函数转换为延续传递风格（CPS）[148]。具体而言，我们提出了一种方法，在抽象语法树层面将一个函数定义拆分为多个函数定义（第4.1.4节），如图4.1（近似地）所示。

4

4.1.4 从命令式函数到数据流

远程函数引用。在第一轮静态分析之后，编译器会使用 Python 类型注解来识别某个函数定义是否引用了远程有状态实体。这些函数可能需要函数拆分。算法会遍历函数定义中的语句，并在发生远程调用或遇到控制流结构时拆分函数。例如，以下 *buy_item* 会调用远程函数*item.update_stock*：

```
1 def buy_item(self, amount: int, item: Item):
2     total_price: int = amount * item.price
3     is_removed: bool = item.update_stock(amount)
4     return total_price
```

该函数在第 3 行的赋值语句处拆分，生成两个新的函数定义：

```
1 def buy_item_0(self, amount: int, item: Item):
2     total_price: int = amount * item.price
3
4
5
6
7
8
9
```

The *buy_item_0* function defines the first part of the original function *and* evaluates the arguments for the remote call. The *buy_item_1* function assumes the remote call *item.update_stock* has been executed, and its return variable is passed as an argument. In general, each function that was split takes as arguments the variables it references in its body and returns the variables it defines. For example, since *buy_item_0* defines the variable *total_price*, its value is returned from the function. Next, since *buy_item_1* uses *total_price*, it is defined as a parameter.

Control Flow. The compiler also needs to split functions when encountering remote function calls within control flow constructs like *if*-statements or *for*-loops. In short, an *if*-statement is split into three new definitions: one that evaluates its condition, one that evaluates the 'true' path, and one that evaluates the 'false' path. Similarly, a *for*-loop is also split into three new definitions: one that evaluates the iterable, one that evaluates the *for*-body path, and one that evaluates the code path after the loop. The function splitting

algorithm is recursively applied to the statements inside the *for* path and inside the true and false path of the *if*-statement.

4.1.5 中间表示

我们的中间表示是一种有状态数据流图，并加入了多个方面的信息。经过静态分析后，每个数据流算子都会补充其可运行的实体/方法名称、输入/返回类型以及方法体。拆分函数后，我们还需要构建一种称为状态机的结构。对于每个拆分后的函数（第 4.1.4 节），我们都会维护一个执行图，用于跟踪给定有状态实体的函数调用所处的执行阶段。

本质上，推导状态机的过程包括展开程序的控制流图。从概念上讲，可以通过推导出有限程序表示来完成向状态机的转换。为此，我们 *i*)不允许无界递归，并且通过为状态机增加额外状态， *ii*) 跟踪循环控制结构的当前迭代次数。调用已拆分的函数时，状态机会被插入函数调用事件中。随着事件流经系统，执行图会被遍历，并调用适当的函数。执行图会存储中间结果——被调用函数的返回值。

4.2 支持的数据流系统运行时

有状态实体可以作为数据流图部署到流式数据流系统中，从而获得恰好一次容错保证。

Flink 的 Statefun。 中间表示会被转换为流式数据流图，例如 Apache Flink 可执行该图。在这种情况下，Kafka 源会将事件推送到入口路由器。入口路由器是一个映射算子，它执行 *keyBy* 操作，将事件路由到正确的有状态映射算子实例，并在该实例中执行函数。每次执行的输出都会转发到出口路由器，再由出口路由器转发到 Kafka 接收器。

我们使用 Kafka 将事件重新插入流式数据流，从而避免循环数据流，因为大多数流处理系统都不支持循环数据流。值得注意的是，我们的系统实现了此进程中路由和执行所需的全部逻辑。另一方面，当事件重新进入数据流以到达拆分函数的下一个函数块时，来自非拆分函数的事件可能引发竞态条件；由于其他事件可能在此期间更改同一函数的状态，进而导致状态不一致。通过水印进行时间跟踪、支持循环数据流以及锁定都可以解决这些问题。由于中间表示与 Statefun 的数据流高度契合，使用 Statefun 运行时只需进行简单的转换和映射。

Styx: a Transactional Dataflow System. Existing dataflow systems cannot execute multi-partition transactions. To this end, we built Styx, a prototype dataflow system in Python. Styx treats each function – and the state effects it creates via calls to other functions – as a transaction with ACID guarantees. We achieve consistency by implementing an extension of Aria [81], a deterministic transaction protocol. The dataflow system is built to allow for dataflow cycles used in function-to-function communication and leverages co-routines for optimal resource utilization. For fault-tolerance, Styx implements the consistent snapshots

protocol [58, 78], which has been adopted by many streaming dataflow systems [8, 9, 149] alongside a replayable source as an ingress, allowing Styx to rollback messages and restore the snapshot upon failure. Although still a prototype, Styx is already able to execute transactional workloads (YCSB-T [127] and partly TPC-C) with promising performance (Section 4.3).

本地。 Styx 数据流图可以在本地环境中执行所有组件。唯一的区别是，状态保存在本地哈希映射数据结构中，而非状态管理后端。本地执行使开发者能够像调试、单元测试和验证任意应用程序一样，对 Stateflow 程序进行调试、单元测试和验证。之后，他们可以将该程序部署到受支持的运行时系统之一。

4.3 初步实验

在本节的实验中，我们选择将 Apache Flink Statefun 与 Styx（第 4.2 节）进行对比。

工作负载。 我们使用了原始 YCSB 基准测试 [126] 中的工作负载 A 和 B。A 是更新密集型工作负载——读取占 50%，更新占 50%；B 是读取密集型工作负载——读取占 95%，更新占 5%。此外，我们还使用 YCSB+T [127]，中的事务性工作负载 T，该工作负载会原子地将一定数量从一个实体的银行账户转入另一个实体的银行账户（2 次读取和 2 次写入）。对于吞吐量测试，我们定义了混合工作负载 M（45% 读取、45% 更新、10% 传输）。对于延迟测试，我们使用齐夫分布和均匀键分布。

Setup. We conducted all the experiments on 14 CPUs: 4 for the Kafka cluster, 6 for the systems, and 4 for the benchmark clients. For Statefun, we gave half of the resources to the Flink cluster and the other to the remote functions. Styx requires a single core coordinator, and the rest are used for its workers.

Baseline. In Styx, we execute complex business logic resulting in state operations. YCSB is a benchmark that supports simple inserts, deletes, and updates, not complete executions of transactions across multiple function calls. It is, therefore, expected that Stateflow, since it executes function calls and application logic, would have a larger overhead than key-value stores. Styx is not a key-value store; instead, it is a stateful function-as-a-service compiler (Stateflow) and runtime that allows programmers to author object-oriented Python code.

延迟。在第一个实验中，我们以较低的 100 每秒请求数，测量了所有 YCSB 工作负载在集成后端系统上的端到端延迟，分别采用齐夫分布和均匀键分布。如图 4.3 所示，两种系统在所有工作负载和分布下的延迟都较低，表现良好。以下是一些值得关注的观察结果。首先，Statefun 在 A、B 两种工作负载以及齐夫分布和均匀分布下的表现相同。这是因为 Statefun 不使用锁定，因此允许并发访问（但也会导致不一致性）。此外，由于所有函数都必须在外部 Python 运行时中运行，读取和写入的成本相同，因为它们都产生网络开销。我们还观察到 Styx 的表现优于 Statefun，因为它支持内部函数间通信，且无需与 Kafka 进行往返通信。请注意，Styx 还支持事务型工作负载，但其延迟高于其他工作负载。不过，如果将一次传输操作视为 2 次读取和 2 次写入操作，则该系统的事务开销很小。最后，我们没有针对 Statefun 运行

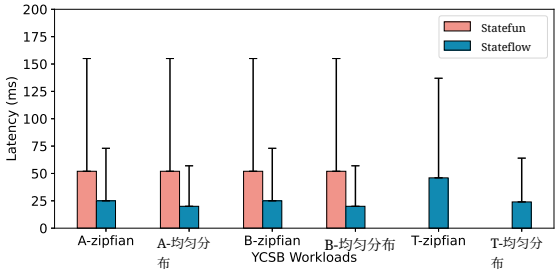


Figure 4.3: Average latency at the 99th percentile, in YCSB (100 RPS) with both Zipfian and uniform key 分布 ns.

事务型工作负载，因为它不支持事务。

吞吐量。在第二个实验中，我们逐渐提高输入吞吐量，并测量端到端延迟。这次，我们使用之前定义的混合工作负载 M（45% 读取、45% 更新、10% 传输）。在图 4.4中，我们观察到的结果与延迟实验一致，直到效率差异开始显现。原因在于，Styx 将执行、状态和消息传递捆绑在一起，因此使用了更多执行核心。相比之下，Statefun 部署在 Apache Flink 集群中将一半 CPU 用于消息传递和状态处理，另一半则用于在远程无状态函数运行时中执行。在当前实验中，就资源利用率而言，这种均衡部署是最优方案。

系统开销。最后，我们还测量了程序转换（函数拆分、插桩等）作为完整运行时一部分所产生的开销（为节省篇幅，未在图中展示）。我们创建了一个合成工作负载，其中不同状态大小从 50 KB 变化到 200 KB。对于每个事件，我们测量了不同运行时组件的耗时。有些组件（如对象构造）归因于程序转换开销，而另一些组件（如状态存储）则归因于运行时。简而言之，函数拆分/插桩仅占总开销的不到 1%。

Conclusion. The experimental evaluation demonstrates the potential of dataflows as an intermediate representation and execution target for scalable cloud applications. In short, these preliminary experiments show that we can translate imperative programs that hide all the aspects of distributed systems and error management from programmers and still achieve high performance. That said, the experiments also uncover the limitations of dataflow systems and implementation issues that we address in the following section.

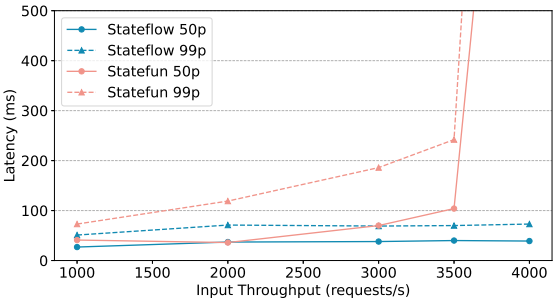


图 4.4：随着输入吞吐量增加，M 工作负载的均值和第99百分位延迟。

4

4.4 开放问题与机遇

The ability to query the global state of a dataflow processor, as well as perform transactional state updates on its state, can transform a dataflow processor into a full-fledged, distributed database system. The envisioned system will be capable of executing Turing-complete "stored procedures" (such as the entity functions in the case of this chapter) that are distributed and partitioned and can perform function-to-function calls with exactly-once guarantees. This is the ultimate goal of this work.

在本节中，我们讨论主要从具有低延迟要求的事务型工作负载中涌现的若干机遇，并概述未来研究方向，以推动数据流系统用于执行通用云应用。

程序分析。 数据流模型本质上是一个 有限状态机，其中节点是原始（图灵完备）程序中的函数，弧表示事件流。对于循环，事件还会携带有关循环前次迭代的信息（例如，循环体和循环条件子句中读取和写入的变量）。这些信息可以正确处理循环（第4.1.5节）。对于方法调用，如果一个方法被映射到单个状态，那么当代码库中存在多个具有不同返回点的调用时，将难以确定调用后应返回的位置。我们将每个方法调用 映射为一次向特定于该调用的状态的转换。这意味着，对同一方法的调用可能会在自动机中产生不同的状态，从而确保每个状态都将正确的返回点作为后继状态。该方法要求对程序进行展开，将运行时可能发生的每个方法调用扩展为不同的状态。

Following this approach, recursive functions would result in a state for each recursive step. Since unbounded recursion would result in infinite automata, we prohibit recursion. Yet, from a compiler perspective, since a program can be CPS-transformed, recursion can be translated into loops via tail-call elimination [150], which could potentially affect the dataflow engine's performance.

此外，在传统上所谓的数据流语言 （例如 Esterel [151],Lucid [152]）中，计算由数据传播驱动——就像在流式数据流中一样。然而，这类语言的表达能力经过有意限制，以便实现高效执行和（自动）验证技术。虽然在这项工作中，我们旨在面向图灵完备的 Python 程序，但表达能力、效率与自动验证之间的权衡仍有待未来研究。

事务。当前的数据流系统保证给定状态键上单事件效果的一致性。为了支持跨有状态实体的事务执行，我们可以采用 单次 事务[121]，或者像我们的原型数据流系统（第 4.2 节）那样，借鉴确定性数据库的理念[81, 101, 124]，以尽量减少事务协调。实践中，很大比例的事务都可以表示为单次事务 [120]；例如，Amazon 的 DynamoDB[122] 和 VoltDB [153] 等广受欢迎的数据库都支持单次事务。这些理念能够定义编程模型如何支持从业者近年来采用的模式，最初包括 SAGAs [91] 和尝试-确认-取消 [96]。

恰好一次、延迟与外部系统。恰好一次保证可能导致较高的延迟：只有在某个纪元成功结束后，数据流的输出才会变得可见[78]。纪元间隔不能太短，否则会带来很高的开销。不过，可以结合因果恢复 [154]、决定因素 [79] 以及可重放接收端，将每个纪元内的延迟降至最低。可重放接收端必须能够检索决定因素。然而，在系统边界处，也就是说，当消息离开数据流图并发送到外部系统时，很难假定接收端支持重放。在这种情况下，应采用更传统的去重技术（例如 HTTP 协议中使用的标准幂等键）。在某些假设条件下（如计算具有确定性、请求队列持久化且可重放等），这类幂等键可以自动生成。不过，对于通用的分布式应用程序而言，情况并非如此；此类应用程序必须自行生成、跟踪、检查并回收唯一标识符，以确保其输出恰好交付一次。在分布式数据库或云编程模型的语境下，这些问题尚未得到充分研究。

Querying Stateful Entities. In previous work [155], we have shown that querying the global state of a dataflow processor can be not only efficient but can also come with certain correctness guarantees. Some work on querying actors has already been done in the context of Orleans [156]. However, querying (e.g., with SQL) a set of entities still poses a number of challenges, especially with respect to the tradeoff between the freshness and consistency of query results. To this end, we could borrow ideas from RAMP (read-atomic) transactions [157] that match well the execution model of transactions and read operations in stateful entities.

4.5 Related Work

The idea of democratizing distributed systems programming is not new. For instance, in [93], the authors mention that a combination of dataflows and reactivity would provide a good execution model for cloud applications. In this work, we share the same belief and build a prototype towards that direction.

Programming models. In the past, approaches like Distributed ML [103], Smalltalk [158], and Erlang [104] aimed at simplifying the programming and deployment of distributed applications. Many of those ideas, including the Actor model, can be reused and extended today. Erlang implemented a flavor of the actor model. Akka [105] offers a low-level programming model for actors. Closest to our work is the Virtual Actors model introduced by Orleans [27, 36], which aims at simplifying Cloud programming and even supports some form of transactions [159]. However, Orleans requires a specialized runtime system

for virtual actors, which does not support exactly-once messaging and does not compile its actors into stateful dataflows.

Imperative programming to Dataflows. The idea of translating imperative code to dataflow is not new. In the database community, there has been work on detecting imperative parts of general applications that can be converted into SQL queries (e.g.,[160]) but also for automatic parallelization of imperative code in multi-core systems. For instance, the work by Gupta and Sohi [161] compiles sequential imperative code to dataflow programs and executes them in parallel. Our work draws inspiration from both these lines of work and extends them by considering the partitioning of state and other considerations that we outline in Section 4.4.

Stateful Functions. A new breed of systems marketed as stateful functions, such as Cloudburst [11], Lightbend's `Cloudstate.io`, and Apache Flink's `Statefun.io` [43], as well as our early prototype in Scala [107], also aim at abstracting away the details of deployment and scalability. However, none of those compiles general-purpose object-oriented code into dataflows.

4.6 Conclusions

In this chapter, we argue that if we want to hide failures from the top-level programming models of Cloud applications, exactly-once guarantees should become first-class citizens. While dataflow systems can provide such guarantees, their programming model makes the development of general Cloud applications cumbersome. To this end, we have developed a compiler pipeline that statically analyzes an object-oriented Python application to create an intermediate representation in the form of a dataflow graph and then submits that dataflow graph to existing dataflow systems. Leveraging dataflow systems' exactly-once guarantees can essentially hide all Cloud failures from programmers with low overhead: our preliminary experimental evaluation demonstrates that function splitting and program transformation incur less than 1% overhead and the YCSB+T benchmark, with low-latency execution.

Current Status. Despite the encouraging results, lots of problems remain open, specifically in the area of transaction execution, programming models, program analysis, and dataflow engines for general cloud applications. Our work currently focuses primarily on *i*) strengthening the formal underpinnings of program transformation to dataflows, *ii*) extending the programming model with different transactional paradigms, and *iii*) further developing Styx, our novel transactional dataflow system.

5

Styx: a Transactional Dataflow-Based Runtime for Stateful Functions as a Service

The previous chapter (Chapter 4) introduced Stateflow, a high-level programming model for cloud applications that compiles object-oriented Python code into distributed dataflows. While Stateflow simplifies application development and abstracts away failures and transactions, it surfaces a set of core runtime requirements to support its execution model—namely, the need for efficient, fault-tolerant, and transactional orchestration of stateful functions.

This chapter presents Styx, a distributed runtime system purpose-built to meet these requirements. Styx implements a novel transactional execution protocol over a streaming dataflow engine, enabling exactly-once semantics and serializable transactions across arbitrary function calls. By integrating deterministic execution, co-located state and compute, and an efficient acknowledgment mechanism, Styx addresses the key limitations identified in existing serverless platforms and transactional SFaaS systems, as discovered in Chapter 3.

The chapter introduces the architecture, programming model, and execution protocol of Styx and evaluates its performance across a range of benchmarks. We conclude the chapter with a demonstration that showcases the system in action, highlighting its ease of use, scalability, and fault tolerance.

K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. Styx: Transactional Stateful Functions on Streaming Dataflows, SIGMOD'25 [15].

K. Psarakis, O. Mraz, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. Styx in Action: Transactional Cloud Applications Made Easy (Demo), VLDB'25 [86].

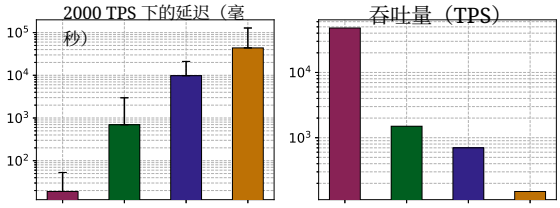


图 5.1：在事务型工作负载中，Styx 的性能至少比最先进水平高出一个数量级（第 5.7 节）。图中展示了延迟的中位数（柱）/99百分位（须）以及吞吐量。对于延迟图，输入吞吐量为每秒 2000 笔事务（TPS）；对于吞吐量图，则报告各系统在亚秒级延迟下实现的吞吐量。

5

Despite the commercial offerings of the Functions-as-a-Service (FaaS) cloud service model, its suitability for low-latency stateful applications with strict consistency requirements, such as payment processing, reservation systems, inventory keeping, and low-latency business workflows, is quite limited. The reason behind this unsuitability is that current FaaS solutions are stateless, relying on external, fault-tolerant data stores (blob stores or databases) for state management. In addition, while multiple frameworks can perform workflow execution (e.g., AWS Step Functions [90], Azure Logic Apps [162]), they do not provide primitives for *transactional* execution of such applications. As a result, distributed applications (e.g., microservice architectures) suffer from serious consistency issues when the responsibility of transaction execution is left to developers [2, 47].

与近期研究 [11–13, 43, 163, 164], 一致，我们认为，要使函数即服务（FaaS）产品成为主流，就应根据有状态函数即服务（SFaaS）范式，为有状态函数提供状态管理支持。此外，我们认为，适用于执行有状态函数工作流的运行时，还应提供 *i*)跨多个函数的端到端可串行化事务保证、 *ii*) 低延迟和高吞吐量执行，以及 *iii*) 高级编程模型，避免使用锁定和事务协调等底层原语。据我们所知，目前没有任何现有方案能够同时满足所有这些要求。

The state-of-the-art transactional SFaaS with serializable guarantees, Boki [12], Beldi [13], and T-Statefun [43] do support transactional end-to-end workflows but induce high commit latency and low throughput. The main reason behind their inefficiency is the separation of state storage and function logic, as well as the use of locking and Two-Phase Commit (2PC) [1] to coordinate and ensure the atomicity of cross-function transactions.

This paper proposes Styx, a novel dataflow-based runtime for SFaaS. Styx ensures that each transaction's state mutations will be reflected once in the system's state, even under failures, retries, or other potential disruptions (known as exactly-once processing). Additionally, Styx can execute arbitrary function orchestrations with end-to-end serializability guarantees, leveraging concepts from deterministic databases to avoid costly 2PCs.

Our work stems from two important observations. First, modern streaming dataflow systems such as Apache Flink [8] guarantee exactly-once processing[8, 78, 79] by trans-

parently handling failures. A limitation of those streaming systems is that they cannot execute general cloud applications such as microservices or guarantee transactional SFaaS orchestrations. Second, deterministic database protocols[80, 81] that can avoid expensive 2PC invocations have not been designed for complex function orchestrations and arbitrary call-graphs. For the needs of transactional SFaaS, Styx leverages a deterministic transactional protocol, enabling early commit replies to clients (i.e., before a snapshot is committed to persistent storage).

我们的工作与该领域近期提出的方案一脉相承，例如 DBOS [92], Hydro [93], 和 SSMS [94]。与这些系统不同，我们采用流式数据流执行模型，并保证跨函数的可串行化性。如图 5.1所示，与最先进水平的可串行化 SFaaS 系统相比，Styx 的中位延迟降低了一个数量级，在 2000 事务/秒时的第 99 百分位延迟降低了两个数量级，吞吐量则提高了一个数量级 [12, 13, 43]。

简而言之，本文作出了以下贡献：
–Styx 将确定性事务与数据流相结合，并克服了这一设计选择带来的挑战（第5.1节）。

– Styx 支持高级 SFaaS 编程模型，可将事务管理和故障管理代码抽象掉（第5.2节）。Styx 通过保证精确一次处理（第 5.3节）以及任意函数调用之间的事务可串行化性（第 5.4节和第 5.5节）来实现这一点。

– Styx extends the concept of deterministic databases to support arbitrary workflows of stateful functions, contributing a novel acknowledgment scheme (Section 5.4.3) to track function completion efficiently, as well as a function-execution caching mechanism (Section 5.5.3) to speed up function re-executions.
– Styx's deterministic execution enables early commit replies: transactions can be reported as committed, even before a snapshot of executed transactions is committed to durable storage (Section 5.5.4).
– Styx outperforms the state-of-the-art [12, 13, 43] by at least one order of magnitude higher throughput in all tested workloads while achieving lower latency and near-linear scalability (Section 5.7).

Styx is available at: https://github.com/delftdata/styx

5.1 Motivation

在本节中，我们分析流式数据流系统设计的具体特征，并论证如何扩展这些系统，使其能够封装一致且高效地执行有状态函数工作流所需的基本原语。我们的工作基于一个关键观察：高性能云服务的架构与并行数据流图非常相似，其中状态经过分区，并与应用逻辑共置 [46]。此外，正如我们在第 5.1.2节中详细说明的那样，确定性事务与数据流系统之间存在协同效应。这种组合能够像过去的单体解决方案一样提供状态一致性并简化编程，同时提升可扩展性并消除开发人员参与。最后，我们展示了如何将确定性事务扩展到 SFaaS；与联机事务处理（OLTP）不同，在 SFaaS 中事务边界是未知的。

5.1.1 Dataflows for Stateful Functions

有状态数据流是几乎所有现代流处理器所采用的执行模型 [8, 143, 165]。除了非常适合并行的数据密集型计算之外，有状态数据流还是支持 Apache Airflow [97], AWS Step Functions [90], 和 Azure Durable Functions[10]等工作流管理器的主要抽象。在下文中，我们将介绍使用有状态数据流构建适用于编排通用云应用程序的运行时的主要动机。

Exactly-once Processing. Message-delivery guarantees are fundamentally hard to deal with in the general case, with the root of the problem being the well-known Byzantine Generals problem [110]. However, in the closed world of dataflow systems, exactly-once processing is possible [8, 78, 79]. As a matter of fact, the APIs of popular streaming dataflow systems, such as Apache Flink, require no error management code (e.g., message retries or duplicate elimination with idempotency IDs).

Co-Location of State and Function. The primary reason streaming dataflow systems can sustain millions of events per second [8, 113] is that their state is partitioned across operators that operate on local state. While the structure of current Cloud offerings favors the disaggregation of storage and computation, we argue that co-locating state and computation is the primary vehicle for high performance and can also be adopted by modern SFaaS runtimes, as opposed to using external databases for state storage.

粗粒度容错。 为确保工作流执行层面的原子性，现有的 SFaaS 系统采用细粒度容错 [12, 13]；每次函数执行都会记录并持久化到共享日志中，然后才调用下一个函数。这要求每次函数调用都与日志记录机制进行一次往返通信，从而显著增加函数执行的延迟。流式数据流系统 [58, 62, 78] 不再记录每次函数执行，而是选择基于异步快照的粗粒度容错机制，以降低这类开销。

5.1.2 确定性与事务

给定一组数据库分区和一组事务，即使发生节点故障或出现并发问题，确定性数据库[101, 124]最终也会达到相同的状态。传统数据库系统提供可串行化保证，允许多个事务并发执行，并确保数据库状态等同于按某种顺序串行执行这些事务所得的状态。确定性数据库不仅保证可串行化性，还保证给定的一组事务即使重新执行，也会对数据库状态产生完全相同的影响。这一保证具有重要意义 [101]，但迄今为止 SFaaS 系统尚未利用这一点。

Deterministic Transactions on Streaming Dataflows. Unlike 2PC, which requires rollbacks in case of failures, deterministic database protocols [80, 81] are "forward-only": once the locking order [80] or read/write set [81] of a batch of transactions has been determined, the transactions are going to be executed and reflected on the database state, without the need to rollback changes. This notion is in line with how dataflow systems operate: events flow through the dataflow graph, from sources to sinks, without stalls for coordination. This match between deterministic databases and the dataflow execution model is the primary motivation behind Styx's design choice to implement a deterministic

transaction protocol on top of a dataflow system.

5.1.3 挑战

尽管数据流系统已取得成功且适用范围广泛，但要将其用于事务性有状态函数，仍需进行多项改造。下文列出了本研究所要解决的挑战和开放问题。

编程模型。目前，数据流系统只能通过函数式编程风格的数据流 API 进行编程：程序员必须重写给定的云应用，使其符合事件驱动的数据流范式。尽管许多应用都可以改写为这种范式，但这需要对程序员进行大量培训，并付出相当大的努力。我们认为，数据流系统若采用面向对象或类 Actor 编程抽象，将更有利于其应用于微服务等通用云应用。

支持事务。在流式数据流系统中，事务通常是指处理一组输入元素及其状态更新，并提供 ACID 保证 [45]。尽管已有进展，仍有一些关键挑战亟待解决，例如多分区事务带来的性能开销，以及锁定和消息重排序所需的数据流阻塞问题。在本研究中，我们认为，要在流式数据流系统中实现事务，就需要通过避免中断数据的自然流动来“让数据持续流动” [166]，同时将事务处理紧密集成到系统的状态管理和容错协议中。

确定性 OLTP 与 SFaaS。采用 Calvin 等确定性协议的 OLTP 数据库，要么需要预先获知每个事务的读写集，要么需要扩展数据库，在首次执行事务时发现其读写集。此外，在这两种场景中，确定性协议都假定事务以单线程函数的形式执行，并且可以从其他分区进行远程读取和写入。对于 SFaaS，任意函数调用使程序员能够兼顾关注点分离原则（该原则在微服务架构中广泛应用）和代码模块化。尽管确定性数据库系统已被证明性能出色，但要为有状态函数的任意工作流设计并实现确定性事务协议并非易事。具体而言，任意函数调用会形成复杂的调用图，而在提交之前必须跟踪这些调用图，以确定事务的边界。

任意工作流执行的数据流。如今，数据流系统的主要用例是流式分析。然而，通用云应用程序有着不同的工作负载需求。函数调用其他函数并接收响应，会在数据流图中引入循环。这些循环可能导致死锁，必须加以处理。

In this work, we tackle these challenges and propose a dataflow system tailored to the needs of stateful functions with built-in support for deterministic transactions and a high-level programming model.

```
1  from styx import Operator
2  from deathstar.operators import Hotel, Flight
3
4  reservation_operator = Operator('reservation', n_partitions=4)
5
6  @reservation_operator.register
7  async def make_reservation(context, flight_id, htl_id, usr_id):
8
9      context.call_async(operator=Hotel,
10                        function_name='reserve_hotel',
11                        key=htl_id)
12     context.call_async(operator=Flight,
13                        function_name='reserve_flight',
14                        key=flight_id)
15
16     reservation = {"fid":flight_id, "hid":htl_id, "uid":usr_id}
17     await context.state.put(reservation)
18
19     return "Reservation Successful"
```

Figure 5.2: Deathstar's [137] Hotel/Flight reservation in Styx. From lines 9-14, the *reserve_hotel* and *reserve_flight* functions are invoked asynchronously. Finally, in lines 16-17, the reservation information is stored. In Styx, the transactional and fault tolerance logic are handled internally.

5

5.2 Programming Model

The programming model of Styx is based on Python and comprises operators that encapsulate partitioned mutable state and functions that operate on that. An example of the programming model of Styx is depicted in Figure 5.2.

5.2.1 编程模型概念

有状态实体。 与面向对象编程中的对象类似，Styx 中的实体负责维护和修改自身的状态。此外，当某个实体需要更新另一个实体的状态时，可以通过函数调用来实现。每个实体都有一个唯一且不可变的键，类似于 Akka 中的 Actor 引用 [35]。，不同之处在于，实体键取决于应用程序，且不包含与其物理位置相关的信息。数据流运行时引擎（第 5.3 节）使用该键，将函数调用路由到负责处理相应实体的算子。

函数。 函数可以修改实体的状态。按照惯例，语境是每次函数调用的第一个参数。函数可以直接调用其他函数，而 Styx 同时支持同步和异步函数调用。例如，在图 5.2的第 9 至 11 行中，已实例化的预订实体将异步调用关联到 Hotel 算子的、键为*"reserve_hotel"*的实体的函数，该实体的键为*"hotel_id"*。类似地，也可以进行同步调用，并阻塞等待结果。在这种情况下，Styx 将阻塞执行，直到调用返回。根据具体用例，可以混合同步和异步调用。不过，异步函数调用还能支持 Styx 在可能的情况下进行进一步优化，详见第 5.4 节和第 5.5 节。

Operators. Each `entity` directly maps to a dataflow operator (also called a vertex) in the dataflow graph. When an *event* enters the dataflow graph, it reaches the operator holding

the *function code* of the given entity as well as the *state* of that entity. In short, a dataflow operator can execute all functions of a given entity and store the state of that entity. Since operators can be partitioned across multiple cluster nodes, each partition stores a set of stateful entities indexed by their unique `key`. When an entity's function is invoked (via an incoming event), the entity's state is retrieved from the local operator state. Then, the function is executed using the arguments found in the incoming event that triggered the call.

状态与命名空间。 如前所述，每个实体只能访问自身的状态。在 Styx 中，状态会根据其所属的实体进行 命名空间划分 。例如，算子 *Hotel* 中的给定键*"hotel53"*表示为：*entities://Hotel/hotel53*。这样，状态对象中给定键的引用便是唯一的，并且在运行时算子跨工作节点分区时可以确定。程序员可以通过 状态上下文 对象调用 context.put() 或 get() 来存储或检索状态（例如，图 5.2的第 5.2.1节）。Styx 的 上下文 与其他系统（如 Flink StateFun 、 AWS Lambda 和 Azure Durable Functions ）中使用的上下文对象类似。

事务。 Styx 中的事务始于客户端请求。事务包含的函数构成一个工作流，并以可串行化保证执行。Styx 的编程模型允许通过引发未捕获异常来中止事务。在图 5.2 的示例中，如果酒店实体在调用 *'reserve_hotel'* 函数时可用量不足，那么 *'make_reservation'* 事务应当中止，同时也应撤销 *'reserve_flight'* 对航班实体所做的潜在状态变更。在这种情况下，程序员必须按如下方式引发异常：

```
...
# Check if there are enough rooms available in the hotel
if available_rooms <= 0:
    raise NotEnoughSpace(f'No rooms in hotel: {context.key}')
...
```

The exception is caught by Styx, which automatically triggers the abort/rollback sequence of the transaction where the exception occurred and sends the user-defined exception message as a reply.

Exactly-once Function Calling.

Styx offers *exactly-once processing* guarantees: it reflects the state changes of a function call execution exactly-once. Thus, programmers do not need to "pollute" their application logic with consistency checks, state rollbacks, timeouts, retries, and idempotency [2, 109]. We detail this capability in Section 5.6.

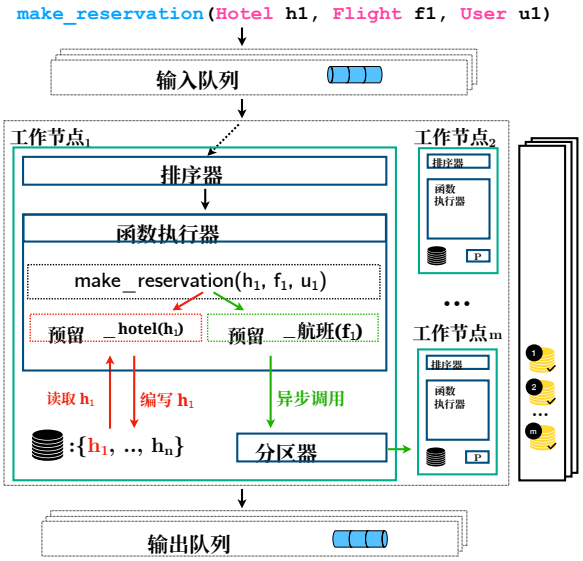


Figure 5.3: Stateful-Function execution in Styx. In each worker, one coroutine manages the sequencing of incoming transactions, while another coroutine handles their processing. In this example, transaction (*make_reservation*) consists of two functions: *reserve_hotel* and *reserve_flight*. A function can access local state (*reserve_hotel*) but also perform remote calls to different partitions (*reserve_flight*). This remote call uses the partitioner to locate the correct worker storing that partition.

5.3 Styx's Architecture

In this section, we describe the components (Figure 5.3) and the main design decisions of Styx.

5.3.1 Components

Coordinator. The coordinator manages and monitors Styx's workers, as well as the runtime state of the cluster (transactional metadata, dataflow state, partition locations, etc.). It also performs scheduling and health monitoring. Styx monitors the cluster's health using a heartbeat mechanism and initiates the fault-tolerance mechanism (Section 5.6) once a worker fails.

Worker. As depicted in Figure 5.3, the worker is the primary component of Styx, processing transactions, receiving or sending remote function calls, and managing state.

The worker consists of two primary coroutines. The first coroutine ingests messages for its assigned partitions from a durable queue and sequences them. The second coroutine receives a set of sequenced transactions and initiates the transaction processing. By utilizing the coroutine execution model, Styx increases its efficiency since the most significant latency factor is waiting for network or state-access calls. Coroutines allow for single-threaded concurrent execution, switching between coroutines when one gets suspended during a network call, allowing others to make progress. Once the network call is completed, the suspended coroutine resumes processing.

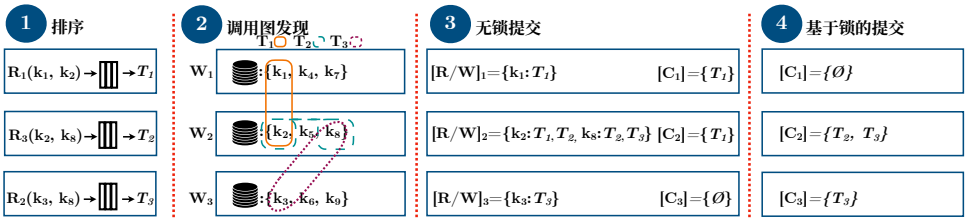


Figure 5.4: The transaction execution pipeline in Styx is divided into 4 parts. First, each external request (R_i) is sequenced as a transaction and is assigned a unique ID. Afterward, the transactions execute their application logic, accessing local keys and performing remote function calls. While a transaction executes, Styx tracks its accessed keys ($[R/W]_i$) and incrementally constructs its call-graph. Subsequently, Styx commits the transactions that do not participate in unresolved conflicts without having to perform locking. For example, we observe that workers W_1 and W_2 are capable to commit $C_1 = C_2 = \{T_1\}$ while T_1 interacts with the same keys as T_2; although it has the lowest id. In the final part, we commit all the transactions by resolving the conflicts with a lock-based mechanism ($C_2 = \{T_2, T_3\}$), $C_3 = \{T_3\}$).

Partitioning Stateful Entities Across Workers. Styx makes use of the entities' *key* to distribute those entities and their state across a number of workers. By default, each worker is assigned a set of keys using hash partitioning.

Input/Output Queue. For fault tolerance, Styx assumes a persistent input queue from which it receives requests from external systems (e.g., from a REST gateway API). Styx requires the input queue to be able to deterministically replay messages based on an offset when a failure occurs. As we detail in Section 5.6, the replayable input queue is necessary for Styx to produce the same sequence of transactions after the recovery is complete and to enable early commit-replies (Section 5.5.4). In the same way, Styx sends the result of a given transaction to an output queue from which an external system (e.g., the same REST gateway API) can receive it. Currently, Styx leverages Apache Kafka [100].

Durable Snapshot Store. Alongside the replayable queue, durable storage is necessary for storing the workers' snapshots. Currently, Styx uses Minio, an open-source S3 clone, to store the incremental snapshots as binary data files.

5.3.2 Transaction Execution Pipeline

Styx 采用基于纪元的事务协议，在每个纪元中并发执行一批事务。一个事务可以包含多个函数，这些函数在运行时会形成由函数调用构成的调用图。每个函数都可以修改其实体状态，而函数调用产生的效果则以事务方式提交到系统状态。在图 5.3 中，一旦*make_reservation* 进入系统，输入队列便会将其持久化并复制。随后，工作节点将该调用接收到本地排序器，由排序器分配事务 ID（TID），并将其中封装的所有函数调用作为一个事务进行处理。在 *make_reservation* 这一情形中，事务由两个函数组成：*reserve_hotel* 和 *reserve_flight*。在此例中，假设*reserve_hotel* 是本地函数调用，而 *reserve_flight* 则在远程工作节点上运行。 *reserve_hotel* 将使用协程以异步方式在本地执行并应用状态变更。相比之下，*reserve_flight* 将在远程工作节点上异步执行，并对远程状态应用变更。

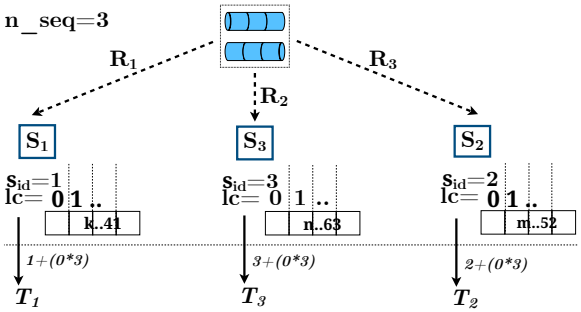


Figure 5.5: Example of TID assignment in Styx with three sequencers. Their identifiers $\{1,2,3\}$ lead to the following sequences: $S_1 = \{1,4,...,k\}$, $S_2 = \{2,5,...,m\}$, $S_3 = \{3,6,...,n\}$ following the formula expressed in Equation (5.1).

5.4 Sequencing & Function Execution

5

要确定性地执行函数并提供可串行化保证，需要经过排序步骤来分配事务 ID（TID）；结合读/写（RW）集合后，该 ID 可用于冲突解决（第 5.5 节）。本节要解决的挑战是确定事务的边界（即事务执行何时开始和结束）；这一问题源于任意函数调用图的执行（第 5.4.3 节）。

5.4.1 Transaction Sequencing

In this section, we discuss the sequencing mechanism (1) of Styx. Deterministic databases ensure the serializable execution of transactions by forming a global sequence. In Calvin [80], the authors propose a partitioned sequencer that retrieves the global sequence by communicating across all partitions, performing a deterministic round-robin.

Eliminating Sequencer Synchronization. Instead of the original sequencer of Calvin that sends $\mathcal{O}(n^2)$ messages for the deterministic round-robin, Styx adopts a method similar to the one followed by Mencius [168], allowing Styx to acquire a global sequence without any communication between the sequencers ($\mathcal{O}(1)$). This is achieved by having each sequencer assign unique transaction identifiers (TIDs) as follows:

$$TID_{sid,lc} = sid + (lc * n_seq) \tag{5.1}$$

其中 $sid \in \mathbb{N}_1$ 是 Styx 协调器在注册阶段分配的排序器标识符，$lc \in \mathbb{N}_0$ 是每个排序器的本地计数器，用于指定其迄今已分配的事务标识符数量，而 $n_seq \in \mathbb{N}_1$ 是 Styx集群中的排序器总数。在图 5.4的示例中，三个工作节点的排序器将分别对 R_1、R_2 和R_3 排序为 T_1、T_3 和 T_2。图 5.5 展示了这些事务标识符如何并行生成。请注意，从概念上讲，Styx 实现了一个分区排序器，其中全局序列 $S = \{S_1 \cup S_2 \cup \cdots \cup S_n\}$是所有分区序列的并集。

缓解序列失衡。 如果单个排序器 S_1 接收的流量多于其他排序器，其本地计数器（lc_1）的增幅就会超过其余排序器的本地计数器。因此，在下一个纪元中，排序器 S_1 生成的事务标识符会大于其他排序器生成的事务标识符。这意味着，到达较空闲排序器的新事务将获得更高的执行优先级：在我们的事务协议中，事务标识符越高，优先级越低。如果工作负载中存在高争用，这会增加繁忙工作节点（S_1）的延迟。为避免这种情况，在每个纪元结束时，协调器会计算最大值 lc（$max(lc_1, lc_2, ..., lc_n)$），并将其传达给所有工作节点，使它们能够调整本地计数器，在每个纪元中重新平衡序列。平衡工作节点的事务优先级可降低第99百分位延迟。

复制与日志记录。 由于输入已记录并复制到可重放队列中，因此 Styx 无需复制和记录排序顺序。发生故障时，事务重放完成后，排序器会生成完全相同的排序顺序（第 5.6.2 节）。

5.4.2 调用图发现

排序后，Styx 需要执行排序后的事务，并确定其调用图和读写集（ ）。为此，函数执行运行时会接收给定纪元中待处理的一系列事务。每个纪元中的事务数量由轮询间隔或可配置的每纪元最大事务数决定（默认每个纪元 1000 笔事务）。我们选择基于纪元的方式，因为分批处理传入事务可以提高吞吐量。

Styx 的运行时会基于数据快照执行所有排序后的事务，以发现读写集。跨越多个工作节点的事务会通过函数调用隐式改变远程工作节点的读写集。发现事务的读写集还存在一个问题：在函数执行之前，事务的调用图尚不明确。这会造成问题，因为协议要求所有事务完成后才能进入下一阶段。为解决这一问题，Styx 提出了一种函数确认方案，详见第 5.4.3 节。

After this phase, all the stateful functions that comprise transactions will have finished execution, and the RW sets will be known. In Figure 5.4, transactions T_1, T_2, and T_3 will execute and create the following RW sets: $Worker_1 \rightarrow \{k_1 : T_1\}$, $Worker_2 \rightarrow \{k_2 : T_1, T_2$ and $k_8 : T_2, T_3\}$ and $Worker_3 \rightarrow \{k_3 : T_3\}$.

5.4.3 Function Execution Acknowledgment

在 SFaaS 范式中，事务形成的调用图是未知的；函数可能由不同的开发团队编写，并且可能形成复杂的调用图。这种不确定性使得难以确定事务何时完成处理，而这至关重要，因为只有在所有事务都完成处理后，阶段 才能开始。为此，为给定事务的每个异步函数调用分配一个 确认_份额。给定函数在运行时统计异步函数调用的数量，从而知道需要创建多少份额。随后，调用函数会向下游函数发送相应的确认份额。例如，在图 5.6中，事务

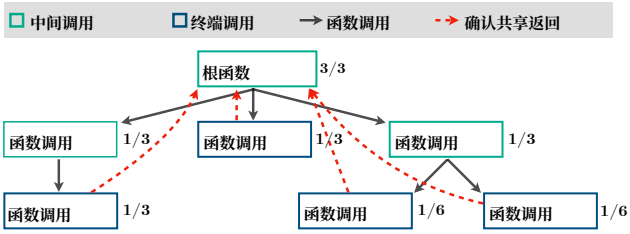


图5.6：异步函数调用链。给定的根函数调用在执行过程中可能会调用其他函数。随着函数执行的推进，原始确认 (3/3) 会拆分成多个部分，每个函数都会收到自己的确认份额。例如，在这次函数执行中，根函数调用了另外三个函数，因此将确认份额拆分成三个相等的部分。后续调用也遵循相同的方式，调用方会进一步拆分自己的确认份额。终端（蓝色）调用（即不再发起其他调用的函数调用）的确认份额之和恰好为 3/3，因此根函数可以报告执行已完成。

入口点（树的根）调用三个远程函数，并将确认_份额拆分为三份（3 × 1/3）。最左侧的函数只调用另一个函数，并将其完整的确认_份额（1/3）传递给它。中间的函数不调用任何函数，因此在执行完成时会将该份额返还给根函数；最右侧的函数调用另外两个函数，并将其份额（1/3）拆分为 2 × 1/6。所有函数调用完成后，根函数应已收集所有份额。当收到的份额总和达到 1 时，根函数/入口点函数便可安全地推断完整事务的执行已经完成。

采用这种设计有两个原因：i）如果每个参与的函数只在完成时发送确认，根函数就无法知道应该等待多少个确认，因而无法判断整个执行是否已经结束；ii）如果使用浮点数而不是分数，就可能遇到浮点数加法相关的问题。例如，在上述包含三个函数调用的示例中，如果使用浮点数，各份额之和不会等于 1，而是 0.99，因为每个份额都是 0.33。由于无法对不精确的除法结果进行准确舍入，因此 Styx 使用分数运算。

A solution close to the *ack_share* is distributed futures [169]. However, it would not work in the SFaaS context as it either requires information about the entire call-graph for it to work asynchronously, or it would need to create a chain of futures that would make it synchronous. Hence, it would introduce high latency for our use case.

5.5 Committing Transactions

After completing an epoch's call-graph discovery, Styx needs to determine which transactions will commit and which will abort based on the transactions' Read/Write (RW) sets and TIDs. To this end, this section presents two different commit phases: *i*) an optimistic lock-free phase that commits only the non-conflicting transactions, and *ii*) a lock-based phase that only commits the transactions that were not able to commit in the first phase. The lock-based commit phase commits all conflicting transactions by acquiring locks in a TID-ordered sequence. To make the second phase faster, we have devised a caching scheme that can reuse the already discovered call-graph to avoid re-executing long function chains

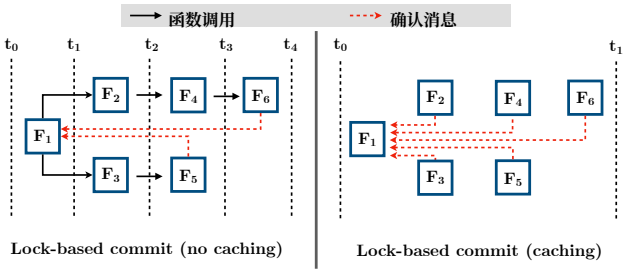


Figure 5.7: If no function caching is performed (left), the transaction execution will execute a deep call-graph; the messages will be sent sequentially and be equal to the number of function calls (5) in addition to the acks (2). Styx's function caching optimization (right) will lead to a concurrent function execution in the lock-based commit phase, between t_0 and t_1, and send only five acks asynchronously.

whenever possible (section 5.5.3).

5.5.1 无锁提交阶段

发生冲突时（即某个事务 t 写入了另一个事务 t' 也在读取或写入的键），与 [81] 类似，只有事务 ID 最小的事务才能成功提交（ ）。尚未提交的事务会被放入队列，以便在下一阶段执行 （保留先前分配的 ID）。

此外，工作节点（W）会通过协调器（$2 * |W|$ 消息）将本地冲突发送给其他所有工作节点：这样，每个工作节点都能掌握所有已中止/重新安排事务的全局视图，并可在本地决定哪些事务可以提交。最后请注意，事务也可能并非因冲突而中止，而是由应用逻辑导致（例如抛出异常）。在这种情况下，Styx 会从读写集中移除相关条目，以进一步减少潜在冲突。

在此阶段，所有未参与冲突的事务都会将其写入应用到状态中，提交事务，并向客户端回复。在图5.4所示的示例中，只有 T_1可以在 W_1 和 W_2 中提交，这是由于 W_2 关于 T_2 和 T_3的读写集存在冲突；更具体地说，冲突发生在键 k_2 和 k_8处。

5.5.2 Lock-based Commit Phase

In the previous phase, 3, only transactions without conflicts can be committed. We now explain how Styx deals with transactions that have not been committed in a given epoch due to conflicts (4). First, Styx acquires locks in a given sequence ordered by transaction ID. Then, it reruns all transactions concurrently since all the read/write sets are known and commits them. However, if a transaction's read/write set changes in this phase, Styx aborts the transaction and recomputes its read/write set in the next epoch. Now, in Figure 5.4, W_2 can sequentially acquire locks for T_2 and T_3, leading to their commits in W_2 and W_3.

5.5.3 Call-Graph Caching

As depicted in Figure 5.4, the lock-based commit phase 4 is used to execute any transactions that did not commit during the lock-free commit phase 3. By the time the lock-based

commit phase starts, the state of the database may have changed since the lock-free commit. As a result, function invocations need to be re-executed to account for the data updates.

在图 5.7左侧，我们描绘了这样一次函数调用。在时间 t_0，F_1 被调用，随后又调用了两个函数链： $F_1 \to F_2 \to F_4 \to F_6$ 和 $F_1 \to F_3 \to F_5$。两个函数链分别在 t_4 和 t_3 完成执行后，便可向根调用 F_1确认其已终止。

缓存的可能性。 在早期实验中，我们注意到一些情况：调用 F_1 时，它调用 F_2 (以及沿$F_1 \to ... \to F_6$ 调用链的后续调用)所使用的参数并未发生变化。这些函数调用的读写集合也同样如此，始终保持不变。由于 Styx 会跟踪这些调用参数以及函数的读写集合，因此它可以在无锁提交阶段缓存输入参数，并在基于锁的提交阶段重新使用，从而避免沿调用链进行耗时的顺序重执行。图 5.7右侧描绘了这种情况：函数调用链不必从 F_1 一直到 F_6按顺序调用，否则会导致较高的延迟。相反，独立工作节点可以在本地并发地重新调用这些函数。这样，所有函数都能并行执行，从而降低延迟和网络开销（见图 $t_4 - t_1$ 5.7）。此外，缓存无需用户输入，对 API 透明，也不依赖于同步或异步规范。不过，在某些条件下，同步调用可以自动转换为异步调用 [55, 170]。

并行函数重新调用的条件。 直观地说，如果调用例如 F_2 时使用的参数和 F_2 的读写集合保持不变，我们就可以放心地假定，函数 F_2 可以并发调用，而无须由 F_1 顺序调用。如果这些函数成功完成，并向根函数 F_1 确认已完成，就意味着事务可以提交。相反，如果任一函数 $F_1 - F_6$ 的读写集合发生变化，或者调用链上任一函数的参数发生变化，就必须重新执行整个事务。在这种情况下，Styx 必须在下一个纪元重新调度该事务。

5.5.4 通过确定性提前回复提交结果

将 Styx 实现为完全确定性的数据流系统，可带来一系列优势，其中包括能够在状态快照持久化到持久化存储之前，就向外部系统（例如客户端）通知事务已提交。传统事务系统只有在 *i*)所请求的事务已提交到持久状态，或 *ii*) 预写日志完成刷新并复制之后，才能向客户端响应。对于 Styx 而言，这意味着必须等到异步快照完成（即持久化到 S3 等持久化存储），从而导致较高的延迟。

Since Styx implements a deterministic transactional protocol that executes an agreed-upon sequence of transactions among the workers, after a failure, the system would run the same transactions with exactly the same effects. This determinism enables Styx to give early commit replies: *the client can receive the reply even before a persistent snapshot is stored.* The assumption here is that the input queue, persisting the client requests, will provide Styx's sequencers with the requests in the same order after replay, a guarantee that is typically provided by most modern message brokers. Performing state mutations

and message passing before persistence has also been explored in DARQ's speculative execution [171].

5.6 Fault Tolerance

Styx implements a coarse-grained fault tolerance mechanism. Instead of logging each function execution, it adopts a variant of existing checkpointing mechanisms used in streaming dataflow systems [58, 78, 79]. Styx asynchronously snapshots state and stores it in a replicated fault-tolerant blob store (e.g., Minio / S3), enabling low-latency function execution. We describe Styx's fault tolerance mechanism below.

5.6.1 Incremental Snapshots & Recovery

The snapshotting mechanism of Styx resembles the approach of many streaming systems[9, 78, 113, 149], that extend the seminal Chandy-Lamport snapshots [58]. Modern stream processing systems checkpoint their state by receiving snapshot barriers at regular time intervals (epochs) decided by the coordinator. In contrast, Styx leverages an important observation: workers do not need to wait for a barrier to enter the system to take a snapshot, since the natural barrier in a transactional epoch-based system like Styx is at the end of a transaction epoch.

快照。为此，Styx 不再通过在系统的各个算子间传播标记来定期创建快照，而是将快照与事务纪元的完成对齐，以便对系统的分布式状态进行一致割，其中包括最新已提交事务的状态、消息代理的偏移量以及排序器计数器（lc）。快照中包含的最少信息为 $O(N+c)$，其中 N 是影响增量映射的更新数量，而 c 是为 Kafka 偏移量和排序器变量存储的固定整数数量。

当快照间隔触发时，Styx 会将当前状态变更复制到一个并行线程中，并异步持久化增量快照，使 Styx 能够在后台执行快照操作的同时继续处理传入事务。快照过程如算法 1 所述。

恢复。发生系统故障时，Styx *i*) 会回滚到最近一次已完成快照所属的纪元，*ii*) 加载快照状态，*iii*) 将可重放源的主题分区（与 Styx 算子分区对齐）回滚到快照时的偏移量，*iv*)加载排序器计数器，最后，*v*) 验证集群运行正常后再执行新的纪元。恢复过程如算法2所述。

增量快照与压缩。每个快照都会以增量映射的形式存储一组状态变更。增量映射是一种哈希表，用于跟踪某个快照间隔内工作节点状态的变更。创建快照时，只对包含当前间隔状态变更的增量映射进行快照处理。为避免跨增量映射跟踪变更，Styx 会定期执行压缩操作，在后台合并增量，如图5.8所示。压缩操作的成本相当于合并两个键空间相同的哈希映射的成本 ($O(N)$)。总成本为 $O(M*N)$，其中 M 表示需要压缩的增量映射数量。

Algorithm 1: Snapshotting Mechanism

Result: Compacted Snapshot stored in durable storage
Input : δ: Delta changes, O_{input}: Input offset, O_{output}: Output offset, E_{count}: Epoch count, SEQ_{count}: 序列计数
Output : $\mathcal{S}$: Compacted snapshot

if *snapshotInterval* **then**
  状态 $\leftarrow \delta$ ▷ Prepare data and metadata for snapshot
  metadata $\leftarrow \{O_{input}, O_{output}, E_{count}, SEQ_{count}\}$
  $\mathcal{S}^{\delta} \leftarrow$ serialize(state, metadata)
  store $\mathcal{S}^{\delta}$
  告知协调器
结束
if *compactionInterval* **then**
  $\mathcal{S} \leftarrow \emptyset$
  foreach $\mathcal{S}^{\delta}$ **do**
    $\mathcal{S} \leftarrow$ compact$(\mathcal{S}, \mathcal{S}^{\delta})$ ▷ Compact delta snapshots
  end
结束

Algorithm 2: Recovery Mechanism

Result: Recovered state from durable storage, possible duplicate messages
Input : $\mathcal{S}$: Latest compacted snapshot,
$\mathcal{S}^{\delta}$: Incremental (delta) snapshots,
O^{last}_{output}: Offset of last output,
Output : $\mathcal{R}$: Set of possible duplicate messages, $state^s$: Snapshotted state, O^s_{input}: Snapshotted input offset, O^s_{output}: Snapshotted output offset, E^s_{count}: Snapshotted epoch count, SEQ^s_{count}: Snapshotted sequencer count

if $\mathcal{S}^{\delta} \neq \emptyset$ **then**
  $\mathcal{S} \leftarrow$ compact$(\mathcal{S}, \mathcal{S}^{\delta})$ ▷ Compact delta snapshots, if any
结束
$state^s, O^s_{input}\ O^s_{output}\ E^s_{count}\ SEQ^s_{count} \leftarrow$ deserialize $\mathcal{S}'$ ▷ Extract persisted state
$R \leftarrow \{m \mid O^s_{\text{output}} \leq m \leq O^{last}_{\text{output}}\}$ ▷ Possible duplicates (Section 5.6.4)

5.6.2 Sequencer Recovery

为确保确定性，在恢复时，Styx 的排序器需要生成与最新快照至故障发生期间所生成的序列完全相同的序列。排序器的恢复协议如下：首先，在创建快照期间，我们会存储每个序列器分区的本地计数器（*lc*）及其标识符（*sid*）和纪元计数器。此外，在每个纪元开始时，Styx 都会记录该纪元中包含的事务数量，即纪元大小。由于 Styx 的纪元大小会变化，且采用了序列器再平衡方案（第5.4.1节），因此需要记录纪元大小。发生故障后，恢复的序列器分区会使用快照中的 *lc* 和 *sid*进行初始化。随后，每个分区都会从其日志中读取自上次快照以来执行的所有纪元的大小。最后，恢复后，排序器会将纪元大小与日志中记录的大小相匹配，从而生成与故障前所观察到的全局序列相同的序列。

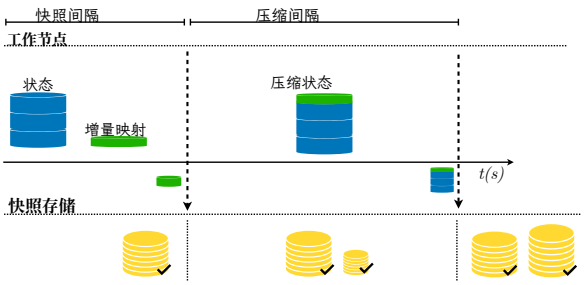


Figure 5.8: Incremental snapshots with Delta Maps in Styx.

5.6.3 Exactly-Once Processing

At first, the durable input queue, which acts as a replayable source, allows Styx to replay requests after failures. By rolling back the queue partitions (aligned with system operator partitions) to the respective offsets as recorded in the latest snapshot, Styx can reprocess only the transactions whose state changes are not reflected yet in the snapshot. Transactions committed and early-commit replies stored in the egress can be deduplicated (Section 5.6.4).

Styx runs each transaction to its completion in a single epoch. A given transaction can execute a large call-graph of functions that can affect the state. If a failure takes place, a transaction's state effects are restored to the latest snapshot, and the complete transaction is re-executed. As a result, no special attention is required to ensure that remote function calls are executed exactly-once, except for resetting all TCP channels between Styx's workers after recovery.

Lemma 1 *The state mutations of committed transactions in Styx are reflected exactly-once, even upon failure.*

Proof 1 *Let S_t denote the state of the system at time t. $Q_t = \{r_1, \ldots, r_n\}$ denotes the durable input queue at time t that holds all requests r_i to be processed. We assume that the input queue operates as FIFO and requests r_i are deterministic. Each r_i will be sequenced as a transaction $T_i = \{upd_l, func_m\}$ by a deterministic sequencer, where upd_l are the state updates and $func_m$ are the function calls of the transaction. We assume that upd_l happens atomically and $func_m$ are also reflected once, given the use of a reliable communication protocol. Given the same initial state S and input from Q, it always produces the same state transition $S \rightarrow S'$, which means $S'_{t+1} = mutation(S_t, Q_t)$. The execution of a transaction T_i is deterministic.*

在任何时刻 t，系统状态 S_t 都反映了 Q_t 中已完全执行并提交的所有事务。因此，状态 S_t 不包含 Q_t中部分执行或正在进行的事务。我们将截至时间 t生成的最新持久化快照记为 $Snapshot(S_t, i, n)$，其中 n 对应于截至时间 t已处理的输入队列中第一个请求 r_i和最后一个请求 r_n 的偏移量。发生故障时，Q_t、$Q_t^{success} = \{r_1, \ldots, r_k\}$的一个子集将包含已成功提交的事务，而子集 $Q_t^{fail} = \{r_{k+1}, \ldots, r_n\}$将包含中止的事务，使得 $Q_t = Q_t^{success} + Q_t^{fail}$。为了从故障中恢复，$Q_t$ 会从 $Snapshot(S_t, i, n)$ 回滚至 S_t，因为我们会持久化输入队列的偏移量。事务 Q_t 会按照原始顺序，从输入队列的偏移量 i 重放到偏移量 n。先进先出队列和确定性排序器确保了这一点。在处理完输入

事务后， $Q_t^{success}$包含已反映在 $Snapshot(S_t)$中的请求，而 Q_t^{fail} 包含待处理请求。由于 $Snapshot(S_t)$ 反映了 $Q_t^{success}$和 $Q_t = Q_t^{success} + Q_t^{fail}$，重放和处理确保了：$S''_{t+1} = mutation(S_t, Q_t^{fail}) = S'_{t+1}$。因此，即使发生故障，所有事务的影响也会精确一次地反映在状态中。

5.6.4 Exactly-Once Output

A common challenge in the fault tolerance of streaming systems is that of the exactly once output [99, 172] in the presence of failures, which is hard to solve for low-latency use cases. For example, in Apache Flink's [8] exactly-once output configuration, clients can only retrieve responses after those are persisted in a snapshot or a transactional sink. This arrangement is sufficient for streaming analytics but not for low-latency transactional workloads, as discussed previously in Section 5.5.4.

To solve that, during recovery, Styx: *i*) reads the last offset of the egress topic, *ii*) compares it with the output offset persisted in the snapshot, determining for which transactions the clients have already received replies, *iii*) retrieves the TIDs attached in those replies, and *iv*) does not send a reply again to the egress topic for those transactions. Note that this deduplication strategy is based on the fact that TIDs have been assigned deterministically.

5.6.5 Addressing Non-Deterministic Functions

As discussed in Section 5.6.1, Styx's recovery mechanism is based on deterministic replay. To this end, Styx requires that the functions authored by developers are also deterministic, i.e., replaying the same function multiple times, using the same inputs and database state, should yield the same results. However, one can achieve determinism even in the presence of non-deterministic logic inside functions, such as randomness (e.g., random numbers/sampling) or calls to external systems (e.g., calling an external database or API). Styx can follow the approach of existing systems (e.g., Temporal [173], Clonos [79]). In the following, we explain how this can be achieved.

Randomness. To retain determinism in the case of randomness, Styx can use an external fault tolerant write-ahead log (WAL) to log the random number along with the TID. Thus, in the case of failure and replay, Styx can use the logged random number, essentially making the function call deterministic during replay.

Calls to External Systems. As illustrated in Figure 5.9, an interaction with an external system needs to consider three critical points to maintain determinism. Styx assumes that the external system supports idempotency [174], meaning that if a call is made twice with the same idempotency key, the effects on the external system's state and its return value will remain the same. In ❶ Styx needs to log the idempotency key and the TID in the WAL before calling the external system. If the external system produces a response (❷), Styx can store it in the WAL and retrieve it from there in case of replay. Finally, when Styx completes a snapshot (❸), it can also clear the WAL for garbage collection since the prior entries are not needed.

Finally, Styx could mask those operations behind an API that exposes the following functionality, such as *system_x.random* for random number generation and *system_x.call_external* for external system calls.

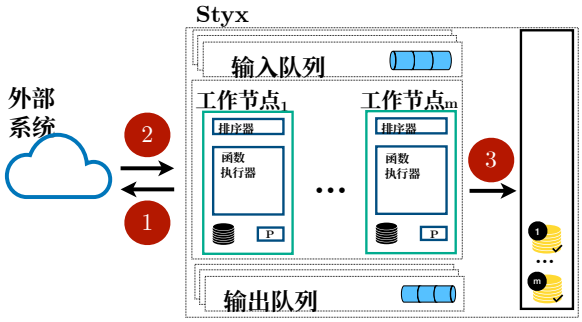


Figure 5.9: External system call critical points and Styx.

5.7 Evaluation

We evaluate Styx by answering the following questions:

- (Section 5.7.2) How does Styx compare to State-of-the-Art serializable transactional SFaaS systems?
- (Section 5.7.2) How does Styx perform under skewed workload?
- (Section 5.7.3) How well does Styx scale?
- (Section 5.7.4) Does the snapshotting mechanism affect performance?

5.7.1 Setup

Systems Under Test. In the evaluation, we include SFaaS systems that provide serializable transactional guarantees. Those are:
Beldi [13]/Boki [12]. Both systems use a variant of two-phase commit and Nightcore [163] as their function runtime and store their data in DynamoDB. Additionally, Boki is deployed with the latest improvements of Halfmoon [175].
T-Statefun [43]. T-Statefun maintains the state and the coordination of the two-phase commit protocol within an Apache Flink cluster and ships the relevant state to remote stateless functions for execution. For fault tolerance, it relies on a RocksDB state backend that performs incremental snapshots.
Styx. Styx is implemented in Python 3.12 and uses coroutines to enable asynchronous concurrent execution. Apache Kafka is used as an ingress/egress and Minio/S3 [176] as a remote persistent store for Styx's incremental snapshots. Finally, Styx is a standalone containerized system that works on top of Docker and Kubernetes for ease of deployment.

Workloads/Benchmarks. Table 5.1 summarizes the three workloads used in the experiments.
YSCB-T [127]. We use a variant of YCSB-T [127] where each transaction consists of two reads and two writes. The concrete scenario is as follows: First, we create 10.000 bank accounts (keys) and perform transactions in which a debtor attempts to transfer credit to a creditor. This transfer is subject to a check on whether the debtor has sufficient credit to fulfill the payment. If not, a rollback needs to be performed. The selection of a relatively small number of keys is deliberate: we want to assess the systems' ability to

场景	#键	函数调用	事务 %
YCSB-T	10k	2	100%
Deathstar 电影	2k	9-10	0%
Deathstar 旅行	2k	3	0.5%
TPC-C	1m-100m	8 / 20-50	100%

Table 5.1: Workload characteristics.

sustain transactions under high contention. In addition, for the experiment depicted in Figure 5.11 (skewed distribution), we select the debtor key based on a uniform distribution and the creditor based on a Zipfian distribution, where we can vary the level of contention by modifying the Zipfian coefficient.

Deathstar [137]。我们采用了 Deathstar [137]，该工作负载由 Beldi 的作者针对 SFaaS 工作负载进行了改编 [13]。它包含两个工作负载：*i*)电影工作负载实现了一个电影评论服务，用户可以在其中撰写电影评论。*ii*) 旅行工作负载实现了一个旅行预订服务，用户可以在其中搜索酒店和航班，按价格／距离／评分排序，查找推荐，并以事务方式预订酒店客房和航班。两个 Deathstar 工作负载都遵循均匀分布。请注意，T-Statefun 无法在这组实验中运行，因为它不支持范围查询。

TPC-C [139]。</style}针对 OLTP 系统的主要事务基准测试是 TPC-C [139]。在评估中，我们使用了 NewOrder 和 Payment 事务，并将它们改写为 SFaaS 范式：将 NewOrder 事务拆分为 20-50 次函数调用（每件商品调用一次），将 Payment 事务拆分为 8 次函数调用。TPC-C 通过增加基准测试中表示的仓库数量来扩展规模／分区数。单个仓库代表偏斜工作负载（所有事务都会访问同一个仓库），而增加仓库数量可以减少争用，从而提高吞吐量并降低延迟。请注意，TPC-C 实验不包含 Beldi、Boki 或 T-Statefun，因为它们不支持 TPC-C。

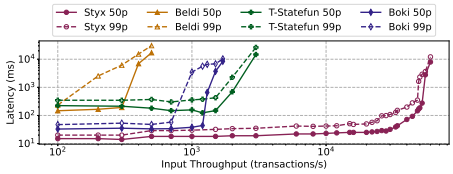
Resources. For Beldi/Boki, T-Statefun and Styx, we assigned a total of 112 CPUs with 2GBs of RAM per CPU, matching what is presented in the original Boki paper [12]. Additionally, throughout all the evaluation scenarios, the data fit in memory across all systems. Unless stated otherwise, Styx and T-Statefun are configured to perform incremental snapshots every 10 seconds. All external systems, i.e., DynamoDB (Beldi, Boki), Minio, and Kafka (Styx, T-Statefun), are configured with three replicas for fault tolerance.

External Systems. Boki and Beldi use a fully managed DynamoDB instance at AWS, which does not state the amount of resources it occupies and is in addition to the 112 CPUs assigned to Boki and Beldi. Similarly, the resources assigned to Minio/S3 (Styx and T-Statefun) are not accounted for.

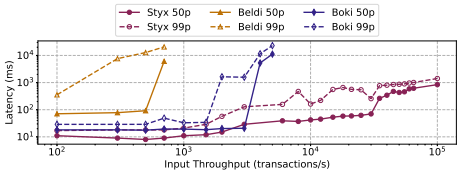
Metrics. Our goal is to observe systems' behavior, measured by their latency, while varying the input throughput.

Input throughput represents the number of transactions submitted per second to the system under test. As the input throughput increases during an experiment, we expect the latency of individual transactions to increase until aborts start to manifest due to contention or high load.

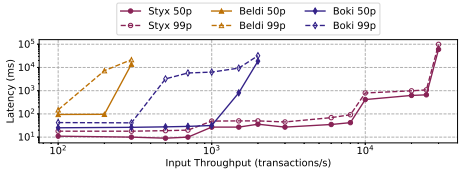
延迟 represents the time interval between submitting a transaction and the reported



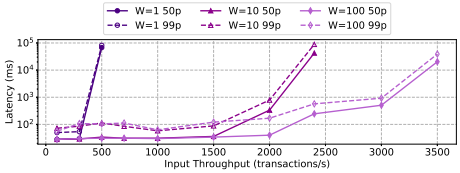
(a) YCSB-T (uniform).



(b) Deathstar Travel Reservation.



(c) Deathstar Movie Review.



(d) TPC-C on Styx with 1, 10, and 100 warehouses.

Figure 5.10: Evaluation in different scenarios. T-Statefun does not support range queries required by the Deathstar workloads. TPC-C is only supported by Styx.

time when the transaction is committed/aborted. In Styx and T-Statefun, the latency timer starts when a transaction is submitted in the input queue (Kafka) and stops when the system reports the transaction as committed/aborted in the output queue. Similarly, in Beldi and Boki, the latency is the time since the input gateway has received a transaction and the time that the gateway reports that the transaction has been committed/aborted.

5.7.2 Latency vs. Throughput

We first study the latency-throughput tradeoff of all systems. We retain the resources given to the systems constant (112 CPUs) while progressively increasing the input throughput. We measure the transaction latency. As depicted in Figure 5.10, Styx outperforms its baseline systems by at least an order of magnitude. Specifically, in YCSB-T (Figure 5.10a), Styx achieves a performance improvement of ~20x in terms of throughput against T-Statefun, which ranks second. In addition, Styx outperforms Boki by ~30x in Deathstar's travel reservation workload (Figure 5.10b) and by ~35x in Deathstar's movie review Figure 5.10c) workload. Finally, in the TPC-C benchmark (Figure 5.10d), which requires a large number of function calls per transaction (20-50), we observe that Styx's performance improves as we increase the input throughput for different numbers of warehouses, reaching up to 3K TPS with sub-second 99th percentile latency (100 warehouses).

中止与吞吐量。Beldi 和 Boki 采用不等待-死亡的并发控制方法，随着吞吐量增加，这会导致大量事务中止。Styx 和 T-Statefun 不使用这种事务中止机制，而是将所有事务执行至完成。在高负载下处理事务的方式不同，使得各系统的延迟难以比较。因此，在图 5.11中，我们绘出了 Styx 和 T-Statefun 的结果，并在单独的表格（表 5.2）中列出 Beldi 和 Boki 的性能及其事务中止率。

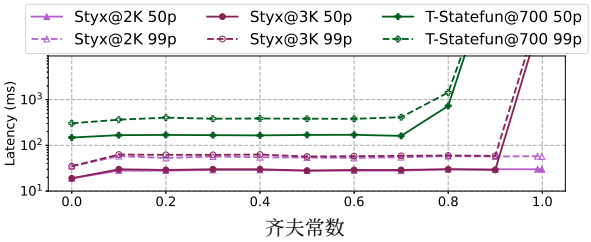


Figure 5.11: Latency evaluation for varying levels of contention (0.0 - 0.999) with YCSB-T (skewed). We ran Styx with two different input throughput variations to show clearly its behavior under contention. Note that Styx and T-Statefun execute all transactions to completion (abort%=0).

		0.0	0.2	0.4	0.6	0.8	0.9	0.99	0.999
Beldi	中止率	47.93	45.54	44.31	47.28	52.40	56.06	61.62	60.70
	CMT 每秒事务数	104	108	111	105	95	76	76	78
Boki	中止率	48.77	48.23	49.54	51.82	61.29	68.50	74.47	70.71
	CMT 每秒事务数	359	362	353	337	271	220	179	205

Table 5.2: Evaluation of Boki and Beldi for varying levels of contention with YCSB-T. We report the abort ratio and committed transactions rate, and omit latency since the systems do not execute all transactions to completion. Both run at their maximum sustainable throughput.

我们观察到以下情况： *i*) 在争用程度最高时（$Zipfian$ 在 0。999），Styx 的吞吐量至少达到每秒 2000 笔事务，其有效吞吐量比其他系统高出 ~5约 10 倍，*ii*) 随着争用程度增加，Beldi 和 Boki（均以最大可持续吞吐量运行）会中止更多事务（~40约 70%），这显著影响了它们的有效性，如表 5.2所示；此外， *iii*) Styx 仅在争用程度较高时才会出现延迟上升（$Zipfian > 0.99$），此时其运行吞吐量仍比其他系统高出 ~4 倍。

Runtime Breakdown. In Table 5.3, we show where the systems under test spend their processing time. We use YCSB-T for this purpose since it is the only benchmark supported by all the systems (Section 5.7.1). We measured the median latency while all the systems were running at 100 TPS for 60 seconds and averaged the proportions of function execution, networking, and state access across all committed transactions. The key observations are: *i*) Styx's co-location of processing and state led to minimal state access latency, and *ii*) Styx's asynchronous networking allows for lower network latency.

要点。 Styx 在所有实验中表现出的显著性能优势，得益于以下三项特性和设计选择： *i*) 如表格所示，将处理与状态共置，并采用高效的网络通信5.3；与其他系统不同，后者必须将状态传输到函数执行引擎； *ii*) 采用带增量映射的异步快照实现容错，相比之下，Beldi/Boki 使用复制，而 T-Statefun 使用基于日志结构合并树的增量快照； *iii*) Styx 采用高效的事务执行协议，而其竞争对手使用两阶段提交。

系统	函数执行	网络	状态访问
Styx	**0.34ms** - 2.2%	**14.33ms** - 95.6%	**0.32ms** - 2.2%
Boki	1.1ms - 3.3%	16.1ms - 49%	15.68ms - 47.7%
T-Statefun	2.76ms - 2.2%	92.12ms - 74.3%	29.11ms - 23.5%
Beldi	1.01ms - 0.7%	56.58ms - 38.4%	89.57ms - 60.9%

Table 5.3: Performance breakdown of all systems. (median latency - percentage from the total)

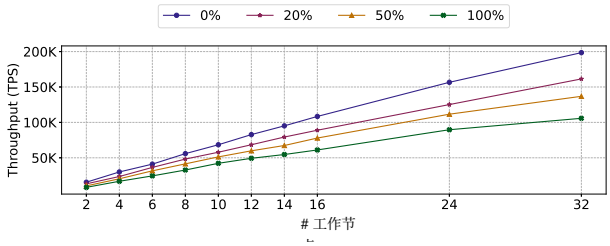


Figure 5.12: Scalability of Styx on YCSB-T with varying percentages of multi-partition transactions.

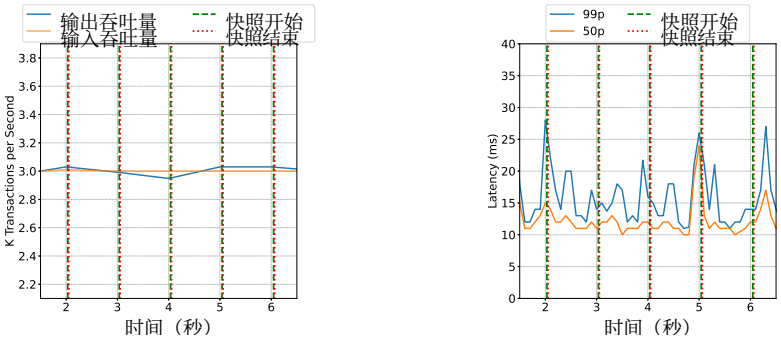
5.7.3 Scalability

在此实验中，我们通过增加 Styx 工作节点的数量来测试 Styx 的可扩展性。每个工作节点分配 1 个 CPU 和包含 100 万个键的状态。我们测量 YCSB-T 上的最大吞吐量。目标是计算当输入吞吐量和工作节点数量同步扩展时，操作的加速比。此外，我们还控制工作负载中多分区事务的比例，即跨越多个工作节点的事务。在图 5.12中可以看到，在所有设置下，Styx 都保持了近乎线性的可扩展性。最后，随着多分区事务数量的增加，Styx 呈现出预期的行为。

5.7.4 Fault-Tolerance Evaluation

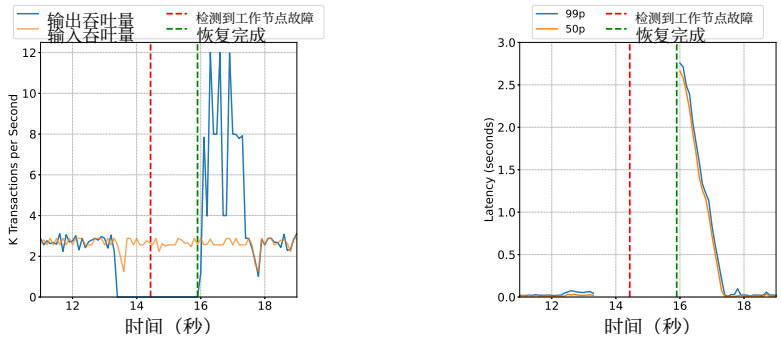
Effect of Snapshots. In Figure 5.13, we depict the impact of the asynchronous incremental snapshots on Styx's performance. In both figures, we mark when a snapshot starts and ends. The state includes 1 million keys, and we use a 1-second snapshot interval. Styx is deployed with four 1-CPU workers, and the input transaction arrival rate is fixed to 3K YCSB-T TPS. In Figure 5.13a, we observe that during a snapshot operation, Styx shows virtually no performance degradation in throughput. In Figure 5.13b, we observe a minor increase in the end-to-end latency in some snapshots. The reason for that is the concurrent snapshotting thread, which competes with the transaction execution thread during snapshotting. At the same time, it also has to block the transaction execution thread momentarily to copy the corresponding operator's state delta.

Recovery Time. In Figure 5.14, we evaluate the recovery process of Styx with the same parameters as in Figure 5.13. We reboot a Styx worker at ~13.5 seconds. It takes Styx's coordinator roughly a second to detect the failure. Then, after the reboot, the coordinator re-registers the worker and notifies all workers to load the last complete snapshot, merge any uncompacted deltas, and use the message broker offsets of that snapshot. The recovery time is also observed in the latency (Figure 5.14b) that is ~2.5 seconds (time to detect the



(a) 吞吐量 (b) 延迟

Figure 5.13: Impact of Styx's snapshotting on performance



(a) 吞吐量 (b) 延迟

Figure 5.14: Styx's behavior during recovery.

failure in addition to the time to complete recovery). In terms of throughput (Figure 5.14a), we observe Styx working on its maximum throughput after recovery completes to keep up with the backlog and the input throughput.

大型状态快照的影响。在图5.15中，我们使用 TPC-C 的 20 GB 大型状态，以及由 100 个单 CPU 工作节点组成、检查点间隔为 10 秒的更大规模 Styx 部署，测试增量快照机制。从 0 秒到 750 秒，Styx 一直在导入数据集。由于没有小增量（导入仅执行追加操作），快照操作比正常工作负载执行开销更大；正常执行时，快照中只存储增量。在 ~550 秒时延迟上升，对应于系统中最大表（Stock 和 Order-Line）的加载。数据加载完毕并以每秒 1000 个事务启动事务性工作负载后，由于增量映射中的状态变更减少，我们观察到延迟有所下降。

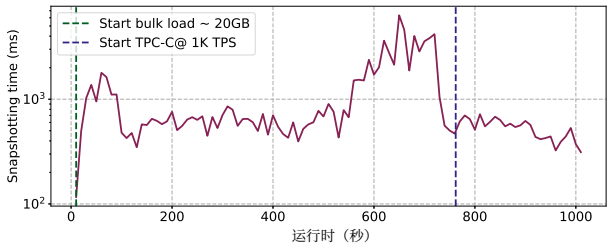


Figure 5.15: Behaviour of incremental snapshots on Styx with ~20GB TPC-C state.

5.8 Related Work

事务性 SFaaS。SFaaS 已受到大量研究关注，并涌现了许多开源项目。要扩大 SFaaS 的普及，必须提供具备容错保证的事务支持（这也是数据库管理系统广受欢迎的原因）。现有系统可分为两类：i）注重事务可串行化；ii）提供最终一致性。第一类包括 Beldi [13], Boki [12], 和 T-Statefun [43]。Beldi 在 DynamoDB 上实现链接式分布式原子亲和日志记录，并通过一种两阶段提交协议的变体，保证 AWS Lambda 函数之间的事务可串行化。Boki 在 Beldi 的基础上改进了事务管道，包括锁定机制和工作流重新执行方面的优化。随后，Halfmoon [175] 在 Boki 的基础上实现了最优日志记录。T-Statefun [43] 同样采用两阶段提交，并通过协调器函数，在 Apache Flink 的 StateFun 之上支持可串行化性。对于最终一致的事务，T-Statefun 实现了 Saga 模式。Cloudburst [11] 还在 DAG 工作流中提供因果一致性保证。较新推出的 Netherite[102] 为微软的 Durable Functions[10], 提供恰好一次保证和高级编程模型，但不保证函数之间的事务可串行化性。Unum [177] 需要与 Beldi 或 Boki 配合使用，才能确保端到端恰好一次和事务保证。

Dataflow Systems. Support for fault-tolerant execution in the cloud with exactly-once guarantees [78, 117] is one of the main drivers behind the wide adoption of modern dataflow systems. However, they lack a general and developer-friendly programming model with support for transactions and a natural way to program function-to-function calls. Closer to the spirit of Styx are Ciel [178] and Noria [82]. Ciel proposes a language and runtime for distributed fault-tolerant computations that can execute control flow. Noria solves the view maintenance problem via a dataflow architecture that can propagate updates to clients quickly, targeting web-based, read-heavy computations. However, neither of the two provides a transactional model for workflows of functions like Styx.

Transactional Protocols. Besides Aria [81] that inspired the protocol we created for Styx Section 5.3, two other protocols fit the requirement of no a priori read/write set knowledge: Starry [179] and Lotus [167]. Starry targets replicated databases with a semi-leader protocol for multi-master transaction processing. At the same time, Lotus [167] focuses on improving the performance of multi-partition workloads using a new methodology called run-to-completion-single-thread (RCST). Styx makes orthogonal contributions to these works and

could adopt multiple ideas from them in the future.

5.9 Future Work

Elasticity in Dataflow Systems. Extensive work has been carried out in dynamic reconfiguration [82–84] and state migration [67–69] of streaming dataflow systems over the last few years. These advancements are necessary for providing serverless elasticity in the case of state and compute collocation to leverage dataflows as an execution model for serverless stateful cloud applications, which is a future goal of Styx.

Replication for High Availability. In the Styx architecture, replication is only applied in the snapshot store and the Input/Output queues to ensure fault tolerance. For high-availability, Styx could adopt replication mechanisms from deterministic databases. Specifically, the design of deterministic transaction protocols, such as Calvin [80], features state replicas that require no explicit synchronization. First, the sequencer replicas need to agree on the order of execution. After that, the deterministic sequencing algorithm guarantees that the resulting state will be the same across partition/worker replicas by all replicas executing state updates in the same order.

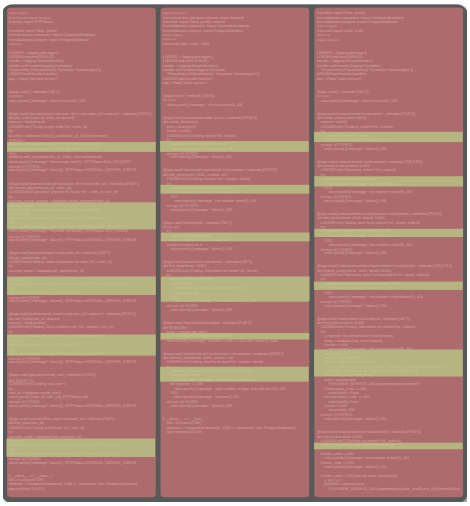
流式数据流中的非确定性函数。 在当前版本中，Styx 要求应用逻辑具有确定性，这与联机事务处理类似 [121, 153], ，在联机事务处理中，存储过程必须具有确定性，因为它们会在不同副本上独立运行。SFaaS [12, 43] 系统也有同样的确定性要求。然而，现实世界的应用程序可能会封装一些逻辑，使其执行结果具有非确定性。非确定性操作的示例包括调用外部系统，以及使用随机数生成器或执行与时间相关的活动。尽管如此，我们计划在 Styx 中支持非确定性函数，具体讨论见第 5.6.5 节。

5.10 结论

本文介绍了 Styx，这是一种分布式流式数据流系统，通过高级标准 Python 编程模型支持具有可串行化隔离保证的多分区事务，并免除了重试和回滚等事务失败管理工作。Styx 遵循确定性数据库范式，同时实现了具有精确一次处理保证的流式数据流执行模型。在所有经过测试的工作负载中，Styx 的吞吐量至少比最先进水平高一个数量级。

5.11 Styx in Action

In this chapter, we introduced Styx [15], a dataflow-based runtime designed for transactional cloud applications built on the aforementioned principles. Styx ensures that each transaction's state mutations are reflected in the system's state exactly-once, even under failures, retries, or other potential disruptions. Additionally, it supports arbitrary function orchestrations with end-to-end serializability by leveraging a deterministic database protocol, eliminating the need for expensive two-phase commits. Our approach is inspired by two key observations [46]. First, modern streaming dataflow systems such as Apache Flink [8] guarantee exactly-once processing by transparently handling failures. However,



(a) Microservice implementation using the saga pattern (Red: code to ensure atomicity and fault tolerance, Green: business 逻辑）。

```
from styx import Operator, StatefulFunction
from shopping_cart.operators import stock, payment, cart
from shopping_cart.exceptions import NotEnoughCredit, NotEnoughStock

@stoc k.reg伊斯特
昱步 def decrement_stock(ctx: 有状态函数 , 数量: int ):
    item_库存 = ctx.get ()
    item_库存 -= 数量
    if item_库存 < 0:
        引发 不足够 hStock(f"Item: {ctx.key} 不 have 足够的库存")
    ctx.put (item_stock)

@payment.reg伊斯特
def pay (上下文: 有状态函数 , 数量: int ):
    credit = ctx.get ()
    credit -= amount
    if credit < 0:
        引发 NotEnoughCredit(f"User: {ctx.key} does 不 h ave enou gh credit")
    ctx.put ( credit )

@购物车.reg伊斯特
def checkout(ctx: 有状态函数 ):
    items, user_id, total_ price , paid = ctx.get ()
    for item_id, qty in items:
        ctx.call_ 昱步 (op运算符=库存 ,
                        函数 _name= 'd递减 _stock',
                        key=item_id,
                        params =(qty, ))
    ctx.call_ 昱步 (op运算符=payment,
                    functio n_name='pay',
                    key=user_id,
                    params=(总计_ price , ))
    paid = True
    ctx.put ((items, 用户 ID, 总价 , 已支付))
    return "Checkout Successful"
```

(b) Checkout workflow in Styx.

Figure 5.16: Comparison between the microservice paradigm (figure 5.16a) and Styx (figure 5.16b).

these systems lack the capability to execute general cloud applications and do not support transactional function orchestrations. Second, efficient transaction execution on top of dataflow systems can be enabled through deterministic database protocols like Calvin [80] or Aria [81] without the overhead of two-phase commits. Styx bridges this gap by integrating a deterministic transactional protocol that allows early commit replies to clients, improving responsiveness while maintaining consistency.

Demonstration Scenarios. To illustrate the capabilities of Styx, in our demonstration, we focus on three scenarios. **(1)** We demonstrate the developer experience by showing how application logic can be free of transaction management and failure handling code. To this end, we have integrated a compiler for transforming object-oriented programs into dataflows optimized for our runtime [55]. **(2)** We highlight the system's deployment and rescaling capabilities, demonstrating how these processes can be performed with minimal overhead. **(3)** We showcase how Styx seamlessly recovers from worker failures without affecting application performance. Additionally, the Styx UI provides live system metrics, offering attendees real-time visibility into system operations.

5.11.1 Demonstration Overview

Scenario 1: Application Development

Figure 5.16 showcases the difference in developing a simplified shopping cart application with three services (stock, payment, cart) between a traditional microservice implementation and Styx. Styx eliminates the boilerplate code needed to ensure ACID guarantees in the microservice implementation and allows developers to focus solely on the core application

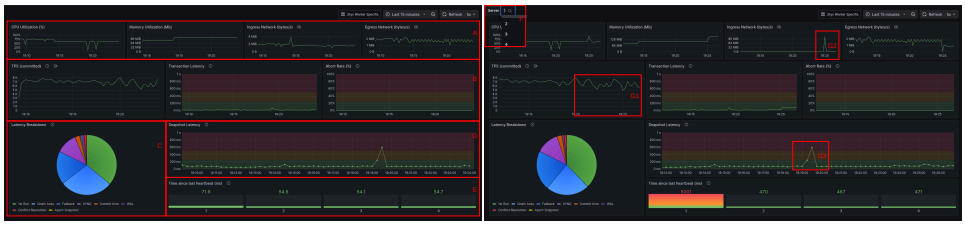


Figure 5.17: Styx monitoring dashboards.

logic. Beyond accelerating development, Styx enhances maintainability and reduces the likelihood of bugs by providing serializable transactions as a service. This results in cleaner, more reliable code, ultimately achieving long-term software quality. Attendees can change parts of the application code and submit applications to the Styx runtime.

Scenario 2: Deployment and Rescaling

Styx is designed for seamless deployment and rescaling. To facilitate real-time monitoring, we have developed two dashboards. In the depicted scenario, we execute the YCSB-T workload across four Styx workers at 10.000 transactions per second (TPS) following a uniform distribution within one million keys.

Part 1: System Overview. Attendees will be able to assess system performance across all Styx workers. The *System Overview* dashboard Figure 5.17 provides a high-level summary of key system metrics:

- **Resource Metrics (A)**: Displays the average CPU and memory utilization, as well as ingress and egress network traffic across Styx workers.
- **Performance Metrics (B)**: Visualizes transaction throughput per second, average transaction latency, and abort rate. Under normal conditions, epoch latency (Styx uses a deterministic epoch-based commit protocol) for the YCSB-T workload remains below 250 ms (green-shaded region). At the same time, the abort rate fluctuates based on the level of contention from 0% (no contention) to 100% (all transactions within an epoch contain the same key at least once).
- **Latency Breakdown (C)**: A pie chart categorizes transaction latency into distinct components. Typically, the primary contributors to latency are the first optimistic transaction execution (1st Run) with the call-graph discovery (Chain Acks), the lock-based fallback commit mechanism (Fallback), and others like cross-worker synchronization, write-ahead-logging (WAL), conflict resolution + commit, and the asynchronous snapshots.
- **Snapshot Latency (D)**: Time taken for a complete delta snapshot throughout the deployment.
- **Worker Health (E)**: The final panel tracks time since the last heartbeat. If this value remains below 1000 ms (green-shaded), it confirms that all workers are healthy and operational.

- **Reconfiguration (G1-3)**: At "19:19:30", we downscale the deployment from four partitions to three, and we observe an increase in snapshot latency and ingress network (data transferred across workers through S3), and finally a decrease in TPS since we decreased the parallelism.

The demonstration attendees will be able to change the skew factors of the supported workloads and perform updates to observe changes in the performance (latency, throughput) of Styx applications in real time.

Part 2: Worker Specific. With a global view of the system's health in mind, attendees can drill down into the performance of individual workers using the *Worker Specific* dashboard. This dashboard mirrors the system overview but focuses on a selected Styx worker. A **dropdown menu (F)** allows attendees to choose a specific worker, enabling direct comparison with the overall system metrics. By analyzing the worker-specific metrics, attendees can quickly pinpoint anomalies. If a single worker exhibits significantly higher transactional latency or reduced throughput compared to the others, it may indicate an overloaded or unhealthy state.

Scenario 3: Fault Tolerance
The final scenario showcases Styx's ability to handle failures efficiently. During the demonstration, attendees can manually terminate a Styx worker to observe how the system detects the failure and triggers its recovery process.

Part 1: System Overview. The system overview dashboard provides a real-time visual indicator of worker failures. When a Styx worker stops responding, the *Time Since Last Heartbeat* metric (panel E) spikes, signaling the loss of communication. This event is accompanied by a sharp increase in transactional latency and a temporary dip in throughput until the system fully recovers. Once the operators assigned to the dead worker are rescheduled to a new or existing worker, Styx begins handling the delayed transactions, and these metrics gradually return to their normal ranges.

Part 2: Worker Specific. Using the worker-specific dashboard, attendees can further investigate the failed worker's behavior. The impact of the failure is more pronounced here. Transaction *latency* and throughput fluctuations become more drastic, and for a brief period, the failed worker will stop reporting metrics entirely. Once the recovery process is completed, these values stabilize, confirming that the system has successfully recovered. This scenario demonstrates Styx's resilience and self-healing mechanisms, ensuring system reliability even in the event of failures.

6

State Migration in Styx: Towards Serverless Transactional Functions

第5章为 *Styx* 奠定了基础，使其成为面向无服务器函数即服务（*SFaaS*）的事务数据流引擎，能够以最少的开发者投入实现强有力的保证和出色的性能。然而，仅凭可编程性和事务性并不能实现无服务器的愿景。真正的无服务器系统还必须提供运维透明性：能够动态扩容和缩容、适应不断变化的负载，并自主重新分配资源，同时保持容错性和正确性。为此，*Styx* 必须超越当前的静态部署模型，采用弹性状态管理。此转变需要一种稳健的状态迁移机制，即使在动态重新配置期间，也能保留 *Styx* 的事务语义和精确一次保证。本章通过介绍 *Styx* 中状态迁移的设计与实施，迎接这一挑战，标志着迈向完全无服务器事务函数的关键一步。

Parts of this chapter are under review:
K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos. State Migration in Styx: Towards Serverless Transactional Functions (Under Review).

Development-wise，Styx 的内部机制对开发者而言早已透明；从这个意义上说，它是“无服务器”的，因为开发者无需编写任何事务或容错代码。接下来的挑战是让其运维方面（例如资源利用率）也变得透明。实现状态迁移机制是迈向这一目标的第一步。与此同时，Styx 继续朝着无服务器系统的方向发展。本章提出对 Styx 的扩展，为其增加状态迁移能力。对于将状态与处理协同 размещ的 Styx 而言，状态迁移是其弹性机制的基石，使其能够提供无服务器服务。如今，Styx 的状态可以按键集粒度分配给不同工作节点，并在工作节点之间迁移。键的迁移有两种方式：对于需要直接访问键的事务，可以按需迁移；对于未访问的键，则可以异步迁移。

本章作出以下贡献：
– Styx 具有弹性，可以在近乎零停机时间内迁移状态，同时保持高吞吐量和低延迟（第6.1节）。
– Styx 量身定制的状态迁移方法优于停止并重启这一基准方案，在扩容和缩容场景下，停机时间减少了 4 倍，并使迁移过程中的事务延迟保持在亚秒级。（第6.4节）。

6.1 Styx 中的状态迁移

在数据流（流处理引擎（SPE）的架构核心）之上实现事务，会给 Styx 的状态迁移支持带来额外挑战。严格针对流处理引擎状态迁移的方法 [67–69] 不适用于我们的使用场景，因为这些方法不管理事务语义。另一方面，面向事务数据库的状态迁移方法 [70, 98, 180] 与传统 OLTP 数据库架构紧密耦合，无法直接应用于 Styx 这样的数据流引擎。因此，Styx 需要一种量身定制的新方法，既能维持事务性，又能很好地适应其恰好一次执行和快照机制。

The most straightforward approach for migrating state in any system is Stop-and-Restart (S&R), where the system will stop processing incoming requests, shuffle the data to their new assignments, and restart processing. More sophisticated approaches often adopt some of the following mechanisms to migrate state: *i*) maintain state replicas across workers to minimize the amount of data in need of migration [68, 69, 180], *ii*) on-demand migration that only sends the data once a worker requires them and [67, 70], *iii*) async migration to transfer data during idle time to progress a migration asynchronously [70, 180].

In Styx, we implement two state migration approaches: a version of S&R tailored to Styx that serves as our baseline method and an approach denoted as Online Migration (OM), which combines elements of migration approaches (*i*) and (*ii*), matching the current state-of-the-art.

In this section, we will first present an overview of state migration in Styx, specify the migration stages (triggering, handling, and resumption of processing), and the changes we made to Styx to support them. Then, we will elaborate on the S&R and OM methods and discuss how we maintain fault-tolerance, determinism, end-to-end exactly-once, and serializability during state migration. In essence, in this work, we aim to extend Styx by adding elasticity, which is the first and most crucial step for Styx to be serverless.

状态迁移变更概述

To support transactional state migration, we extend Styx's base core runtime. This section outlines the architectural modifications required to enable this capability. We assume that migration is initiated by an external client or Styx's coordinator based on metrics such as load imbalance or resource utilization. The migration triggering policy, including autoscaling heuristics and monitoring, is considered orthogonal and left as future work. To enable correct and efficient state movement, Styx introduces the following system-level changes:

分区级状态与元数据。 为了让状态迁移更加灵活，算子状态和 Kafka 偏移量采用更细粒度的方式进行跟踪，具体来说，是按分区跟踪。不考虑状态迁移时，偏移量和状态都是按算子维护的，因此无法区分单个工作节点上同一算子的多个分区。更细粒度的跟踪支持有选择地迁移状态，无需依靠哈希来确定键所属的分区，也不必创建完整的算子级检查点。

影子分区。 Kafka 消息的到达时间无法保证与 Styx 内部的重新配置同步。为应对这种潜在的不同步，Styx 会暂时保留影子分区，将分区外事务转发到正确的分区。这样，Styx 无需全局暂停输入即可确保正确性。缩容时这一问题尤为明显，因为缩容会移除分区。例如，客户端发起缩容迁移操作并触发重新分区时，其他客户端可能仍会根据先前的分区方案向 Styx 发送事务，导致事务到达已不复存在的过时分区。将旧分区保留为影子分区，由其专门转发此类事务且不承担任何状态变更职责，这对于 Styx 保持精确一次处理保证至关重要。

Global Offset Restoration. A rerouting mechanism is required, not only during downscaling involving the shadow partitions but as a part of the general migration solution. Its role is twofold: *i*) to detect incoming transactions from the input queue, routed to an outdated partition due to client partitioning misalignment, and *ii*) reroute them to the correct partition. Following our previous example, a transaction can be routed to an outdated partition until all of Styx's clients update their routing table and align with the new partitioning scheme. To maintain exactly-once processing and output, it is essential to restore the input/output queue offsets, as they might get updated in at most two places (previous and new partitioning). This ensures that no records are skipped or reprocessed during migration in case of failure.

Blocking Actions Minimization. To ensure low latency even in the presence of large data transfers during migration, we had to make the two following adjustments to Styx *i*) add compression to large messages and *ii*) streaming asynchronous snapshots. First, Styx enforces compression using the Zstandard compression algorithm [181] for messages larger than a configurable size (by default set to 1MB). Second, regarding the snapshot mechanism in Styx, Styx now spawns a background thread that receives state deltas in a streaming fashion to prevent blocking if the delta becomes large (i.e., under heavy load, the involved subset of keys is significantly large). In the previous version, Styx would

Algorithm 3: Stop and Restart

Input : P_w: Current partitions assigned to a worker, $H^{new}_{w_i}$: New hash function per partition, K: Keys, w_{id}: Worker id, O_{in}: Input offset, O_{out}: Output offset, E_{count}: Epoch count, SEQ_{count}: 序列计数
Output: G: new Dataflow Graph

foreach $w_i \in workers$ **do**
  $P^{new} \leftarrow \emptyset$ ▷ Snapshotted partitions
  对于每个 $P_{w_i} \in P_w$ **do**
    foreach $(key, value) \in P_{w_i}$ **do**
      $new_p \leftarrow H^{new}_{w_i}(key)$
      $P^{new}_{new_p} \leftarrow P^{new}_{new_p} \cup (key, value)$
    end
  end
  Store P^{new} to persistent storage
  $meta_{new} \leftarrow \{O_{in}, O_{out}, E_{count}, SEQ_{count}\}$
  发送 $meta_{new}$ 转至协调器
结束
$G \leftarrow$ NewDataflowGraph(())
foreach $w_i \in workers$ **do**
  $subG_{w_i} \leftarrow$ AssignSubgraph((G, w_i))
  接收 $meta_{new}$ 来自协调器
  $O_{in}, O_{out}, E_{count}, SEQ_{count} \leftarrow meta_{new}$
结束
foreach $w_i \in workers$ **do**
  foreach $P^{new}_i \in P^{new}$ **do**
    $P_i \leftarrow P^{new}_i$ ▷ 恢复分区
  end
结束

accumulate the entire delta and then send it to the background thread, leading to significant latency spikes that are now resolved.

Composite Key Partitioning. In its current version, Styx also supports composite key partitioning to enhance data locality. Keys can be grouped by logical attributes (e.g., *warehouse_id* in TPC-C that is a prefix in all table primary keys other than the *item*), allowing transactions to access colocated partitions. During migration, Styx leverages this structure to colocate groups of related keys to the same worker. This optimization reduces cross-worker communication and improves transaction commit latency.

These design extensions allow Styx to support both synchronous (stop-and-restart) and asynchronous (online) migration strategies without violating transactional or fault tolerance guarantees while maintaining low latency during reconfiguration.

6.2 State Migration Methods

6.2.1 Stop & Restart

Stop-and-Restart (S&R) is the most straightforward migration strategy that we could implement on top of Styx. It suspends execution, performs state migration, and resumes computation with updated routing. Styx implements an optimal variant of S&R tailored to its transactional runtime. In Algorithm 3, we detail S&R where, at first, for each worker



Figure 6.1: Alignment of epoch phases with online state migration. In phase 2 transactions determine their call-graphs, and therefore, the associated keys need to be migrated immediately. Contrarily, in 3 and 4 keys can be migrated asynchronously without interfering with the transactional protocol.

Algorithm 4: Online Migration

Input : P_w: Current partitions assigned to a worker, H_i^{new}: New hash function per partition, K: Keys, w_{id}: Worker id, O_{in}: Input offset, O_{out}: Output offset, E_{count}: Epoch count, SEQ_{count}: 序列计数

Output: G: new Dataflow Graph

foreach $w_i \in workers$ **do**
  foreach $P_i \in P_w$ **do**
    foreach $key \in P_i$ **do**
      $new_p \leftarrow H_i^{new}(key)$
      $P_{new_p}^{new} \leftarrow P_{new_p}^{new} \cup (key)$
    end
  end
  $meta_{cur} \leftarrow \{O_{in}, O_{out}, E_{count}, SEQ_{count}, P_i^{new}\}$
  Send $meta_{cur}$ to Coordinator
结束
$G \leftarrow$ NewDataflowGraph(())
foreach $w_i \in workers$ **do**
  接收 $meta_{new}$ 来自协调器
  $O_{in}, O_{out}, E_{count}, SEQ_{count}, P_i^{new} \leftarrow meta_{new}$
  $subG_{w_i} \leftarrow$ AssignSubgraph((G, w_i))
结束

(w_i) and all the partitions assigned to them (P_{w_i}) Styx rehashes all the keys of that partition based on the new partitioning using its hash function ($H_{w_i}^{new}$) and adds them alongside their values to the new partitions (P_{new}). Once a worker finishes the hashing step, the new partitions are stored as a snapshot of the persistent storage. Then, each worker sends their metadata (input/output offsets, sequencer count, and epoch count) to the coordinator, concluding the 'Stop' step. To 'Restart' Styx based on the new partitioning, each worker is assigned its part from the graph and receives the updated metadata from the coordinator. Finally, the worker loads the new partitions from the previously rehashed snapshot stored in persistent storage.

While simple, robust, and independent from the transactional protocol, S&R incurs downtime due to the rehashing, storing, and loading of the data. This violates availability, making it more suitable for planned migration settings than Styx's serverless requirements.

6.2.2 Online

To support online, near-zero-downtime migration, Styx introduces an online method adapted to its transactional model. In contrast to S&R, the online method performs migra-

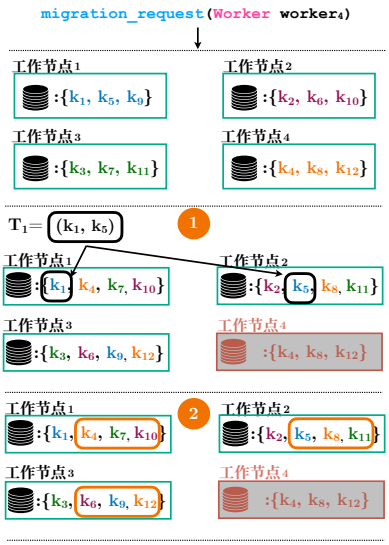


Figure 6.2: State distribution before and after migration request, which removes $worker_4$. After migration, we see the example of a transaction ($T_1 = \{k_1, k_5\}$). T_1 interacting with keys k_1 and k_5, meaning that while k_1 is already in-place, k_5 needs to be migrated on-demand as migration phase 1 suggests in Figure 6.1. The rest of the keys, assuming that there is no other transaction interacting with them, can be migrated asynchronously in the migration phase 2.

6

tion in an on-demand and asynchronous manner, allowing the system to remain available throughout. Styx leverages its transactional epoch protocol to piggyback migration steps on normal processing cycles. As shown in Figure 6.1, keys accessed during Phase 2 are migrated synchronously and on-demand to ensure consistency. Other keys, if idle, are migrated asynchronously during subsequent phases using worker idle time and batching mechanisms in phases 3 and 4. This asynchronous migration is essential for the migration to complete, which is necessary for the fault tolerance mechanism to be reactivated (snapshots are switched off mid-migration as per the SotA approaches to maintain consistent snapshots).

在算法4中，我们详细介绍在线迁移方法。与 S&R 类似，对于每个工作节点（w_i）及分配给它的所有分区（P_{w_i}），Styx 会使用其哈希函数（H_i^{new}），根据新的分区方案对该分区中的所有键重新计算哈希值。核心区别在于，它不会添加值，而是会创建路由表，记录键所在的位置及其目的地（P^{new}）。工作节点完成哈希步骤后，会将这些信息连同元数据（输入/输出偏移量、序列器计数和纪元计数）发送给协调器。最后，工作节点加载新的路由表和元数据，并结合事务协议，使用图
和 所示的按需（1）和异步（2）机制执行迁移。在图6.2中，我们展示了 Styx 如何在重新分区的同时执行缩容操作：工作节点数量从4 个降至3 个，并演示了按需迁移和异步迁移。

6.3 Maintaining Guarantees

这两种状态迁移方法都需要维护 Styx 的正确性保证，即： *i*)精确一次处理、 *ii*) 确定性、 *iii*) 可串行化事务保证以及 *iv*)恰好一次输出。对于 S&R 方法，维持正确性很简单，因为它会停止执行、重新分配数据，然后重启。这不会影响 Styx 中任何已就位的机制，这些机制已在第5.6节详细介绍。因此，本小节将主要说明在线迁移方法如何在维护 Styx 正确性保证的同时运行。

Styx maintains deterministic execution and guarantees serializability throughout online migration. When a transaction requires access to a key located on another worker (triggering On-Demand Migration), the worker blocks execution until that key is received. This procedure is safe, as Styx's single-process coroutine approach ensures that no other transaction on the same worker can simultaneously request the same key. Transactions can only operate on fully available and up-to-date keys, and migrations are aligned with epochs to ensure consistency. Additionally, the asynchronous phase of Online Migration is only performed after the call-graph of all transactions within the epoch has been discovered. At that point, all the requested key transfers of the on-demand migration phase are guaranteed to have been completed. Moreover, fault tolerance remains unaffected; if a failure occurs, Styx will recover from the latest snapshot and restart the migration without compromising correctness. For the same reason, exactly-once processing and output remain unaffected by the migration mechanism.

Finally, the only critical point to be addressed in both the S&R and Online migration methods is out-of-partition events due to client-server partitioning misalignment. In Section 6.1, we explained the two new mechanisms of Styx that address this issue, namely Shadow Partitions and centralized offset restoration. Shadow partitions are used temporarily to reroute out-of-partition transactions from Kafka, ensuring that the correct worker and partition process the incoming transaction. To fully address this issue, the Kafka offset progress that might be affected by two different workers is restored by the coordinator before being stored in a snapshot. This coordination ensures exactly-once processing in the case of failure during state migration.

6.4 State Migration Experiments on Styx

In this section, we evaluate the state migration mechanism of Styx in sustainable throughput for scaling up and down while repartitioning the entire state. The repartitioning operation involves considerable data movement since Styx's partitioner is hash-based.

设置

Styx executes in Python 3.13 and contains the optimizations mentioned in Section 6.1.

Workload. The workloads used in our experiments follows the SotA transactional approaches [70, 180], which are the YCSB and TPC-C benchmarks. In this experiment section, all tables are partitioned into 16 parts.

YSCB [126]. The Yahoo! Cloud Serving Benchmark is a suite of workloads designed to represent large-scale, commonly developed web services. In our experiments, we use two YCSB datasets: a smaller dataset with 1 million records (1GB) for small-state experiments

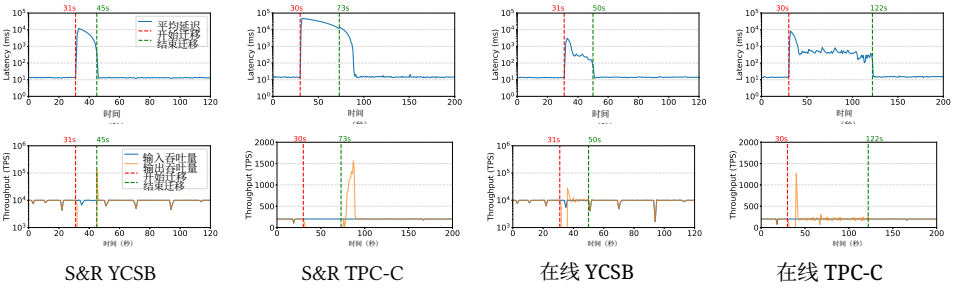


Figure 6.3: Stop and Restart (S&R) and Online Migration for big State (10GB) Scale Down in both TPC-C and YCSB: Latency (top row) and Throughput (bottom row).

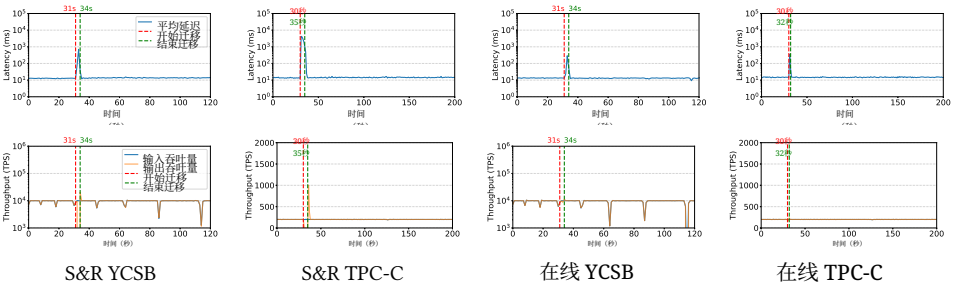


Figure 6.4: Small State 1GB Scale Down: Latency (top row) and Throughput (bottom row).

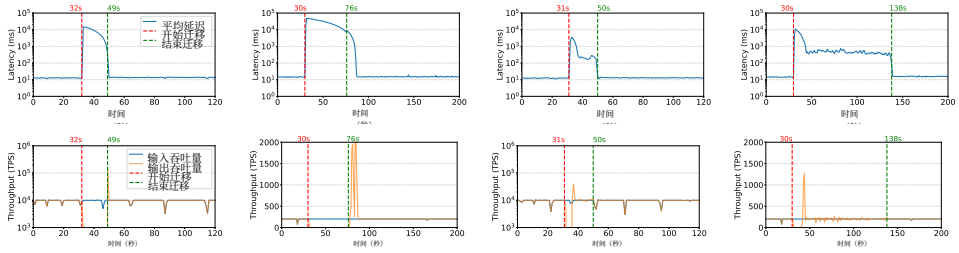
and a larger one with 10 million records (10GB). Each record consists of a primary key and 10 columns containing 100-byte randomly generated strings. We follow the workload configuration from [70], which includes two transaction types: 15% of operations perform a single-record update, while the remaining 85% perform a single-record read. It is important to note that YCSB differs from the YCSB-T variant used in Section 5.7.

TPC-C. We follow the exact spec of TPC-C as in Section 5.7.1 and generate two datasets, a small one (10 warehouses, 1GB) and a larger one (100 warehouses, 10GB) for our experiments in this section.

Metrics. In all the migration experiments, we measure input/output throughput, mean latency, and the migration interval.

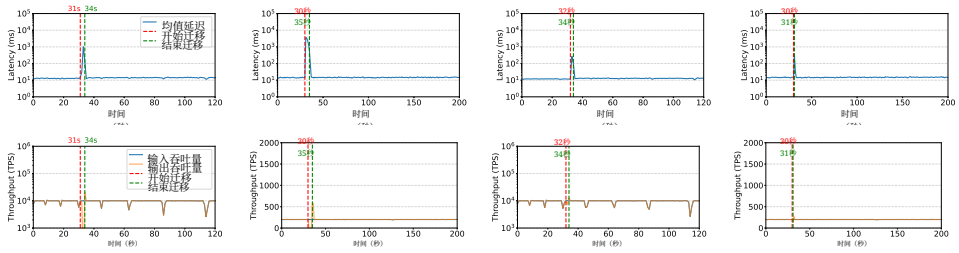
Input/Output Throughput. In addition to *input throughput* as mentioned in Section 5.7.1, we also display the output throughput, which is the number of transaction responses Styx produces per second. During migration, we expect *i*) the input throughput to remain stable since we do not pause the clients and *ii*) the output throughput to drop.

Mean Latency. For the state migration experiments, latency is defined in the same way as in Section 5.7.1. The only deviation from our previous latency reporting is that in line with prior work, [67, 70, 182], we report mean latency instead of the 99th percentile (P99). This decision is motivated by the observation that, during migration, the system enters a transitional phase in which latency spikes are the norm. While P99 latency metrics effectively capture worst-case performance under steady-state conditions, they tend to produce inflated values during migration, which can obscure a clear understanding of



S&R YCSB S&R TPC-C 在线 YCSB 在线 TPC-C

Figure 6.5: Big State 10 GB Scale Up: Latency (top row) and Throughput (bottom row).



S&R YCSB S&R TPC-C 在线 YCSB 在线 TPC-C

图 6.6：小状态 1GB 扩容：延迟（顶行）和吞吐量（底行）。

6

system behavior. In this context, mean latency provides a more informative measure of the overall impact of migration on transaction performance.

迁移间隔。一个重要的仅迁移指标是迁移间隔，即迁移进程所需的时间。为展示这一点，我们绘制了所有实验中的迁移开始和结束时间戳。

结果

我们在大状态和小状态下，针对扩容和缩容场景运行了 YCSB 和 TPC-C 工作负载。

缩容。 在缩容场景中，我们将 16 个 Styx 单 CPU 工作节点减少到 12 个，并将状态重新分区为相同数量的分区。在图 6.3 中，我们观察到，在 YCSB 和 TPC-C 中，S&R 方法均受到极高延迟（数十秒）的影响；在迁移和重新分区 10 GB 数据期间，YCSB 的停机时间为 14 秒，TPC-C 的停机时间为 43 秒。在线方法在两个工作负载中都表现出轻微的延迟升高，这与迁移阶段开始时的重新哈希操作有关，且延迟升高均不超过 10 秒；停机时间也很短，YCSB 接近 5 秒，TPC-C 接近 10 秒。在 YCSB 中，无论状态大小，小状态和大状态的迁移耗时大致相同，S&R 方法略占优势。相反，在 TPC-C 中，在线方法耗时是其两倍，因为 TPC-C 包含更多需要传输的键，并且异步迁移配置为每个事务周期传输 5,000 个键。

在 Figure 6.4 with the smaller state, we observe the same trends, but the migration

impact is much smaller.

扩容。 在扩容场景中，我们将 Styx 工作节点从 12 个 1 CPU 工作节点增加到 16 个，同时将状态重新分区为相同数量的分区。在展示大状态和小状态扩容实验的图 6.5 和 6.6 中，我们观察到与缩容实验类似的行为，这在意料之中，因为这两种迁移方法都不受扩容/缩容语义的影响。唯一的区别是迁移耗时更长，因为缩容时由 16 个线程执行初始重新哈希阶段，而扩容时则由 12 个线程执行。

要点。 总体而言，在线迁移方法在所有关键性能指标上都表现更优，例如 *i*) 停机时间：在线迁移至少比停止并重启快 4 倍； *ii*) 峰值平均延迟不超过 10 秒；而且，一旦计算出哈希值且 Styx 追上输入，事务延迟便会立即降至 1 秒以下。需要注意的是，在线方法唯一落后的指标是迁移时间。之所以这一点很重要，是因为迁移期间会关闭容错机制，而且发生故障时的回滚窗口更长。

6.5 Related Work

State Migration. Squall [70] performs live state migration in transactional systems by locking involved partitions using a dedicated special transaction. It supports on-demand data movement but relies on DBMS-level deadlock handling and assumes range partitioning for optimizations. Clay [98] incrementally migrates frequently co-accessed keys using a cost model to balance data movement and transaction performance during the migration. Albatross [183] focuses on migrating state in shared storage systems by incrementally copying in-memory cache and active transaction state. To maintain consistency, Albatross relies on two-phase commit. Meces [67] supports fine-grained, on-demand state migration targeting stream processing systems using markers. While ensuring exactly-once semantics, transactions are out-of-scope. Rhino [68] targets query reconfiguration on stream processing systems, maintaining state replicas and utilizing virtual nodes without transaction support.

自动扩缩容。 其他研究则针对流处理引擎中的动态重新配置。DS2 [83]是一种基于控制的自动扩缩容方案，利用算子的到达率和处理率来确定何时需要扩缩容。该方案专注于流处理引擎的动态重新配置，并利用流查询的拓扑结构，一次性扩缩容所有算子。为此，它会逐步计算每个算子的最优并行度。Dhalion [84] 是一个基于控制的框架，使用反压机制进行速率控制。它会监控负载倾斜和慢实例等反压信号，以检测资源争用，并据此触发扩缩容操作。DRS [184] 是一种基于排队论的自动扩缩容器，旨在捕捉已配置资源的影响。它基于稳态假设，使用排队论模型对算子行为建模，相较于依赖反压或到达率，为延迟估算提供了结构更完善的框架。最后，水平 Pod 自动扩缩器（HPA）[185] 是 Kubernetes 附带的默认自动扩缩容方案。HPA 会对部署进行水平扩缩容，依据观测指标使其达到用户提供的目标值；观测指标可由用户定义（例如，平均 CPU/内存利用率）。然而，Styx 并未处理触发自动扩缩容的因素，这部分留待未来研究。

7

讨论与结论

本论文研究了事务型云应用的问题空间。尽管已有少数系统解决了该问题空间的部分问题，且各自具备优势与局限性，但仍存在重大挑战。我们确定了五个关键研究空白，并以此为工作提供动机：*i*) 缺乏适用于事务型云应用的有原则的抽象和底层支撑，*ii*)先前系统的局限性使其难以胜任通用运行时，*iii*)开发正确且可扩展的事务型应用十分困难，*iv*) 在确保强事务保证和容错保证的同时实现高性能面临挑战，以及 *v*) 需要在动态工作负载下实现高效且自适应的执行。本章首先结合已确定的研究空白，总结本论文的主要发现与贡献。随后，我们将反思本研究更广泛的影响，并探讨以本论文奠定的基础为依托开展未来研究的方向。

7.1 Main Findings

Dataflows as a Substrate for Transactional Cloud Applications

In Chapter 2, we presented our reasoning behind the selection of a dataflow engine as a suitable substrate for transactional cloud applications, considering the following research question:

> **RQ-1:** What would be the optimal substrate for a system serving complex cloud applications?

To answer **RQ-1**, we analyzed the requirements of modern cloud applications from a developer's perspective and observed that event-driven microservices share structural similarities with stateful dataflow graphs. This led us to propose dataflow engines as a promising foundation for cloud runtimes, due to their ability to model asynchronous communication, stateful processing, and parallelism. This theoretical foundation guided the practical investigations in subsequent chapters.

The Limitations of Prior Approaches

To verify our reasoning, in Chapter 3 we enhanced Apache Flink Statefun, an existing SFaaS system, with transactional guarantees and named it T-Statefun. This was led by the following research question:

> **RQ-2:** Is it possible to use an existing SFaaS dataflow system for this purpose? If so, what are the limitations?

T-Statefun answered **RQ-2** and demonstrated that it is indeed feasible to retrofit a stream processing system with transactional capabilities. However, our findings revealed key limitations. Although T-Statefun outperformed existing SFaaS systems, its serializable protocol showed poor scalability under high-contention workloads, and its reliance on remote state access increased latency. Additionally, the system's API complexity made it difficult for non-expert developers to use it effectively, underscoring the need for better programmability.

编程难题

In Chapter 4, we tackled the programmability issue by introducing *Stateflow*, a domain-specific language embedded in Python that allows developers to write object-oriented cloud applications using familiar imperative constructs. This work addressed the following research question:

> **RQ-3:** Can we design a domain-specific language that works on top of all stream processing systems, creating a simple and easy-to-use object-oriented API?

To answer **RQ-3**, we designed Stateflow as a DSL that compiles Python programs into a dataflow intermediate representation (IR). This abstraction decoupled application logic from the underlying execution engine, enabling the abstraction to be seamlessly compiled against different stream processors. By focusing on Python, we reduced the entry barrier for developers, enabling them to build complex transactional workflows without having to reason about concurrency or distributed systems internals.

Simple Highly Performant Transactional Cloud Applications

In Chapter 5, we addressed the limitations in performance and flexibility of existing systems by designing and building Styx, a high-performance distributed dataflow runtime tailored to transactional SFaaS. The following research question drove this work:

> **RQ-4:** Can we build a system that enables developers to write transactional, data-intensive cloud applications without requiring expertise in distributed systems?

To answer **RQ-4**, we developed Styx, a streaming runtime that natively supports Stateflow applications with end-to-end transactional serializability and exactly-once guarantees. Styx achieved high throughput and low latency by applying forward-only deterministic transaction processing and avoiding expensive two-phase commits. It executed arbitrary orchestrations of function calls while preserving consistency and leveraging the dataflow execution model to enable parallelism and fault-tolerance. This work demonstrated that transactional guarantees and performance do not have to be mutually exclusive in cloud application runtimes.

Adaptivity and Efficiency

Finally, in Chapter 6, we explored the dynamic scalability of Styx by incorporating elasticity and adaptivity, guided by the following research question:

> **RQ-5:** Can we give Styx elasticity properties, such as state migration, allowing it to become serverless?

To address **RQ-5**, we extended Styx with a state migration mechanism that enables live relocation of operator state while preserving correctness. We adapted techniques from prior work on stream processor reconfiguration to work within the context of transactional function orchestrations. Our approach enabled Styx to scale applications elastically and recover from load imbalance without downtime, paving the way for a serverless execution model that supports long-lived, stateful cloud applications with fine-grained resource management.

7.2 Limitations

Despite the contributions of this thesis, there are some limitations we would like to address.

高可用性。 如第 5章所述，Styx 旨在通过确定性执行、输入消息重放和定期快照，高效地从故障中恢复。这使其能够快速进行故障恢复，并保证恰好一次语义。然而，Styx 目前尚不支持高可用性（HA）部署，即故障转移可以瞬时完成，用户几乎察觉不到停机时间。在当前形态下，发生故障时，受影响的算子必须从持久化存储中恢复状态并重放输入日志，这会引入恢复延迟。要实现高可用性，需要采用诸如主动-备用复制之类的机制，使每个算子都拥有一个接收相同输入流并保持状态最新的副本。发生故障时，该副本可以立即接管，从而无需从磁盘重新加载状态并重放消息。

与可重放消息队列的松耦合。 Styx 目前假设输入流通过 Apache Kafka 接入，而 Kafka 同时充当入口和出口机制，也是两个持久性层之一。Kafka 强大的顺序保证和可重放性简化了 Styx 的容错模型，使其能够通过消息重放实现确定性恢复。然而，这种设计也使 Styx 与 Kafka 的传递语义紧密耦合，可能限制部署灵活性，并妨碍与其他消息系统无缝集成。具体而言，Styx 假设输入队列的分区方式与其内部并行度相匹配，以便高效排序并实现确定性执行。这些假设主要出于性能和简洁性的考虑，但并非保证正确性所必需。若要取消这些假设，就需要重新考虑如何在各输入队列之间保证因果顺序、分区对齐和恰好一次语义。总体而言，当前以 Kafka 为中心的设计受益于其成熟度和稳健性，但若能支持与更多弹性或无服务器入口层兼容，将提升 Styx 的部署可移植性，并使其更适用于多样化的云环境。

Analytical Queries. Styx is purpose-built for transactional workloads that require low latency, fine-grained updates, and strong consistency guarantees. It does not currently support analytical queries or Hybrid Transactional and Analytical Processing (HTAP)

workloads, which aim to unify real-time decision-making with historical insight. This limits Styx's applicability in scenarios where the latest transactional state must be queried or analyzed in conjunction with larger historical datasets, for instance, in fraud detection, personalized recommendations, or operational dashboards. Supporting HTAP workloads in a system like Styx is a non-trivial task. A key challenge lies in balancing the freshness of analytical results with the latency and throughput of transactional processing. Styx could explore integration with analytical backends or log-based data lake ingestion to expose transactional state externally, while maintaining its streaming semantics. Another approach could be the introduction of read-only views with bounded staleness guarantees or side-channel query engines that operate on compacted snapshots. Designing such hybrid execution models while preserving Styx's exactly-once guarantees, deterministic behavior, and low overhead remains a direction for future work. However, it is beyond the scope of this thesis.

Limited Adaptivity Capabilities. Although Styx supports elasticity through manual operator scaling and state migration, it currently lacks fully automated autoscaling mechanisms that are a requirement of modern serverless platforms. Users must manually provision and assign workers, and the system does not yet include dynamic operator placement, load-aware partitioning, or resource-aware scaling. Consequently, Styx cannot automatically respond to changes in workload intensity, nor can it scale to zero during periods of inactivity. This limits its ability to provide cost-efficient and responsive execution, especially in cloud-native and pay-as-you-go settings [186].

Nonetheless, the architectural choices in Styx, particularly its modular operator model, logical dataflow execution, and separation between control and data planes, lay a promising foundation for implementing adaptive, serverless behavior. For example, operator placement decisions could be informed by runtime statistics or predictive models, while idle operators could be garbage-collected and rehydrated using persisted state. These enhancements would move Styx toward a fully serverless execution model, where resource management becomes transparent, and users can focus entirely on application logic.

Exploring these capabilities remains outside the scope of this thesis but represents a direction for future research. Integrating reactive scaling policies would enable Styx to support workloads with highly variable demands, thereby creating a general-purpose, transactional serverless dataflow engine.

Long-Running Transactions. Styx currently assumes that transactions are short-lived and bounded in both time and size. As a result, it does not support long-running transactions that can potentially span multiple input epochs. Because the system employs epoch-based processing with coordination barriers, a single long-running transaction can delay the entire epoch's progression, resulting in increased latency and reduced system throughput.

This limitation reflects a broader tension between isolation and liveness in distributed systems. Supporting long-running transactions with strict serializability often requires locking or coordination mechanisms that severely impact performance and availability. In response to this, alternative models such as SAGAs have been proposed [91], which decompose a long transaction into a series of smaller, compensatable steps. These approaches trade strong consistency guarantees for better scalability and resilience, particularly in cloud-native applications [96].

For many real-world applications, strict isolation may not be necessary. For exam-

ple, external API calls, human-in-the-loop workflows, or machine learning-based fraud detection can take seconds or even minutes to complete. In these cases, developers may prefer looser guarantees (e.g., eventual consistency, compensation mechanisms) to maintain responsiveness. Enabling support for such workloads would require a redesign of Styx's execution model, likely through relaxed isolation semantics, compensation patterns, or asynchronous orchestration techniques, which remains an open area for future exploration.

Styx Core Written in Python. Python enables rapid prototyping and lowers the barrier to entry for building distributed systems like Styx. However, it lacks low-level control over memory management, task scheduling, and high-performance networking. These limitations restrict the ability to fine-tune execution performance and concurrency behavior, especially in production environments. While the architectural principles and abstractions of Styx are language-agnostic, implementing the runtime in a systems programming language like Rust or C++ could significantly improve throughput and latency. Languages like Rust, in particular, offer compile-time guarantees that help prevent entire classes of concurrency-related bugs, making it a promising candidate for future reimplementation of the core dataflow engine.

Moreover, user applications in Styx do not need to be written in the same language as the runtime. With the growing maturity of WebAssembly (WASM), it becomes feasible to support applications authored in multiple languages that compile to a common execution target. This could expand Styx's reach by decoupling the user-facing API from the runtime implementation, paving the way for a multi-language, safer, and more performant execution model.

7.3 Future Directions

In this section, based on the insights and experience we gained from developing Stateflow and Styx, presented in this thesis, we identified open challenges regarding Stateflow/Styx and the field in general. Guided by these, we briefly discuss potential future directions.

Styx

Auto-Scaling. Future work could implement autoscaling mechanisms that adjust operator parallelism in response to workload characteristics. This includes both horizontal scaling (adding/removing workers) and shuffling based on key-based access patterns, ideally without requiring user intervention or manual tuning.

Fault-Tolerant State Migration. State migration in existing systems is often performed with fault tolerance mechanisms temporarily disabled, limiting their applicability in highly available environments. A promising direction is the inception of new online, fault-tolerant migration techniques that enable state transfer between workers without pausing or compromising fault tolerance. Moreover, such approaches could open the door to stacked migrations, where new migration plans can be initiated before prior ones have fully completed, enabling faster and more adaptive reconfiguration.

Analytical Workloads for Styx. Expanding Styx to support analytical workloads would bridge the gap between real-time transactional processing and decision-making, enabling a unified platform for Hybrid Transactional and Analytical Processing (HTAP). This integra-

tion is essential for modern cloud-native applications, which increasingly require the ability to react to operational events with strict transactional guarantees while simultaneously supporting low-latency analytical queries on fresh data.

要在 Styx 中实现 HTAP，需要进行若干系统级增强。首先，在现有的基于行或实体的执行模型之外引入列式存储引擎，将使 Styx 能够高效地存储和扫描大量历史数据。集成 HTAP 能力也会给资源管理和迁移带来新的挑战。例如，状态迁移机制必须考虑分析查询局部性——迁移的不仅是热点键，还包括分析物化结果或汇总数据，并将其迁移至最需要它们的位置。这进一步凸显了对轻量、非干扰式在线迁移协议的需求，这类协议既能保持容错能力，也能确保查询一致性。

长时间运行事务的协议。目前，Styx 支持的事务协议基于纪元，这意味着在某个纪元中的所有事务完成之前，不能开始新的事务。长时间运行的事务会阻止纪元完成，导致后续所有纪元的延迟显著增加。为了支持长时间运行的交互，未来工作可以探索专门用于管理多纪元事务或长时间运行事务的协议。这可能包括补偿机制、Saga 或分布式协调策略，在不阻碍进展的同时保持可串行化性。

Stateflow

Complete Domain-Specific Language for Cloud Applications. While Stateflow demonstrates the viability of a Python-embedded DSL for cloud dataflows, future work could formalize a standalone domain-specific language that captures application semantics declaratively. Such a language would provide static analysis, performance optimization, and correctness guarantees that exceed what Python currently offers.

7

基准测试

事务型云应用的基准测试。要改进现有系统并优化新系统，准确反映真实应用的基准测试至关重要。然而，业界缺少用于评估事务型云原生应用的标准化基准测试。设计一套具有代表性的基准测试套件，涵盖真实的工作负载、服务级别协议和云部署模型，将有助于为 Styx 等系统提供背景，并促进不同架构之间进行有意义的比较。

参考文献

参考文献

[1] J. N. Gray. 关于数据库操作系统的笔记。《操作系统：高级课程》，1978年。[2] Rodrigo Laigner、Yongluan Zhou、Marcos Antonio Vaz Salles、Yijian Liu 和 Marcos Kalinowski。微服务中的数据管理：实践现状、挑战与研究方向。*VLDB*基金会会刊，14(13):3348–3361，2021年9月。[3] Christos H Papadimitriou。并发数据库更新的可串行化性。《*ACM*会刊》（*JACM*），26(4):631–653，1979年。[4] 亚马逊 Amazon Lambda。https://aws.amazon.com/lambda/。[5] 谷歌云函数。https://cloud.google.com/functions。[6] 微软 Azure Functions。https://azure.microsoft.com/en-us/products/functions。[7] Jim Gray 和 Andreas Reuter。事务处理：概念与技术。爱思唯尔，1992年。[8] Paris Carbone、Asterios Katsifodimos、Stephan Ewen、Volker Markl、Seif Haridi 和 Kostas Tzoumas。Apache Flink：单一引擎中的流式处理与批处理。*IEEE*计算机学会数据工程技术委员会通讯，36(4)，2015年。[9] Michael Armbrust、Tathagata Das、Joseph Torres、Burak Yavuz、Shixiong Zhu、Reynold Xin、Ali Ghodsi、Ion Stoica 和 Matei Zaharia。结构化流处理：Apache Spark 中用于实时应用的声明式 API。载于*SIGMOD*，2018年。[10] Sebastian Burckhardt、Chris Gillum、David Justo、Konstantinos Kallas、Connor McMahon 和 Christopher S. Meiklejohn。Durable Functions：有状态无服务器的语义。《*ACM*编程语言汇刊》，5(OOPSLA)，2021年10月。[11] Vikram Sreekanti、Chenggang Wu、Xiayue Charles Lin、Johann Schleier-Smith、Jose M Faleiro、Joseph E Gonzalez、Joseph M Hellerstein 和 Alexey Tumanov。Cloudburst：有状态函数即服务。*arXiv*预印本 *arXiv:2001.04592*，2020年。[12] Zhipeng Jia 和 Emmett Witchel。Boki：基于共享日志的有状态无服务器计算。载于*ACM SIGOPS*第二十八届操作系统原理研讨会论文集，第691–707页，2021年。

[13] Haoran Zhang、Adney Cardoza、Peter Baile Chen、Sebastian Angel 和 Vincent Liu。容错事务型有状态无服务器工作流。载于第 *14* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），2020年。[14] Martijn de Heus、Kyriakos Psarakis、Marios Fragkoulis 和 Asterios Katsifodimos。利用有状态数据流实现跨无服务器函数的事务。《信息系统》，108:102015，2022年。[15] Kyriakos Psarakis、George Christodoulou、George Siachamis、Marios Fragkoulis 和 Asterios Katsifodimos。Styx：流式数据流上的事务型有状态函数。《*ACM* 数据管理会议论文集》，2025年。[16] S. Srinivasan。云计算演进，第1–16页。Springer New York，纽约，纽约州，2014年。[17] R. P. Goldberg。虚拟机架构。载于虚拟计算机系统研讨会论文集，第74–112页，纽约，纽约州，美国，1973年。计算机协会。[18] J.E. Smith 和 Ravi Nair。虚拟机架构。*Computer*，38(5):32–38，2005年。[19] Bryan Ford、Mike Hibler、Jay Lepreau、Patrick Tullmann、Godmar Back 和 Stephen Clawson。微内核与递归虚拟机相遇。载于Karin Petersen和Willy Zwaenepoel（编），第二届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集，美国华盛顿州西雅图，*1996*年*10*月*28*日至*31*日，第137–151页。ACM，1996年。[20] Rajdeep Dua、A Reddy Raja 和 Dharmesh Kakadia。虚拟化与容器化：支持平台即服务。载于*2014*年 *IEEE* 云工程国际会议，第610–614页，2014年。[21] Emiliano Casalicchio。容器编排：综述，第221–235页。Springer International Publishing，Cham，2019年。[22] Swaminathan Sivasubramanian。Amazon DynamoDB：可无缝扩展的非关系型数据库服务。载于*2012*年*ACM SIGMOD*国际数据管理会议，第729–730页，2012年。[23] Bradley Barnhart、Marc Brooker、Daniil Chinenkov、Tony Hooper、Jihoun Im、Prakash Chandra Jha、Tim Kraska、Ashok Kurakula、Alexey Kuznetsov、Grant Mcalister、Arjun Muthukrishnan、Aravinthan Narayanan、Douglas Terry、Bhuvan Urgaonkar 和 Jiaming Yan。Aurora Serverless中的资源管理。*VLDB*基金会会刊，17(12):4038–4050，2024年。[24] Ram Kesavan、David Gay、Daniel Thevessen、Jimit Shah 和 C. Mohan。Firestore：面向应用开发者的 NoSQL 无服务器数据库。载于第*39*届 *IEEE* 国际数据工程会议（*ICDE 2023*），美国加利福尼亚州阿纳海姆，*2023*年*4*月*3*日至*7*日，第3376–3388页。IEEE，2023年。

[25]Apache Flink Statefun。有状态函数：独立于平台的有状态无服务器技术栈。https://nightlies.apache.org/flink/flink-statefun-docs-stable 访问日期：2025年4月5日。[26]Azure Durable Functions。
h
t
t
p
s://
l
e
a
r
n.
microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview，2018年。[27] Phil Bernstein、Sergey Bykov、Alan Geller、Gabriel Kliot 和 Jorgen Thelin。Orleans：用于实现可编程性和可扩展性的分布式虚拟参与者。*MSRTR2014*，41，2014年。[28] Henri E. Bal、Jennifer G. Steiner 和 Andrew S. Tanenbaum。分布式计算系统的编程语言。《*ACM*计算调查》，21(3):261–322，1989年9月。[29] Peter Alvaro、Tyson Condie、Neil Conway、Khaled Elmeleegy、Joseph M Hellerstein 和 Russell Sears。Boom analytics：探索面向云的数据中心声明式编程。载于*EuroSys*，2010年。[30] Yuan Yu、Michael Isard、Dennis Fetterly、Mihai Budiu、Úlfar Erlingsson、Pradeep Kumar Gunda 和 Jon Currey。DryadLINQ：使用高级语言进行通用分布式数据并行计算的系统。载于第八届 *USENIX* 操作系统设计与实现大会论文集，OSDI'08，第1–14页，美国，2008年。USENIX 协会。[31] Fan Yang、J. Shanmugasundaram、M. Riedewald 和 J. Gehrke。Hilda：面向数据驱动 Web 应用的高级语言。载于第*22*届国际数据工程会议（*ICDE'06*），第32–32页，2006年。[32] Henri E. Bal、M. Frans Kaashoek 和 Andrew S. Tanenbaum。Orca：用于分布式系统并行编程的语言。*IEEE*软件工程汇刊，18(3月):190–205，1992年。[33] Java Spring。https://spring.io 访问日期：2025年4月5日。[34] Python Flask。https://flask.palletsprojects.com/en/stable访问日期：2025年4月5日。[35] Akka.io。https://akka.io/。[36] Sergey Bykov、Alan Geller、Gabriel Kliot、James R Larus、Ravi Pandya 和 Jorgen Thelin。Orleans：人人可用的云计算。载于*SoCC*，2011年。[37] Dan Pritchett。Base：ACID的替代方案。载于文件系统与存储，第6卷。ACM Queue，2008年。[38] Chuzhe Tang、Zhaoguo Wang、Xiaodong Zhang、Qianmian Yu、Binyu Zang、Haibing Guan 和 Haibo Chen。Web应用中的临时事务：好的一面、坏的一面与丑陋的一面。载于*2022*年国际数据管理会议论文集，SIGMOD '22，第4–18页，纽约，纽约州，美国，2022年。计算机协会。

[39] Gul Agha。《*Actors*：分布式系统中的并发计算模型》。麻省理工学院出版社，1986年。[40] Dapr 分布式应用运行时。 https://dapr.io 访问日期：2025年4月5日。[41] Zijun Li、Linsong Guo、Jiagan Cheng、Quan Chen、Bingsheng He 和 Minyi Guo。《无服务器计算综述：设计架构技术入门指南》。《*ACM*计算调查》，54(10s)，2022年9月。[42] Dmitrii Ustiugov、Plamen Petrov、Marios Kogias、Edouard Bugnion 和 Boris Grot。《无服务器函数快照的基准测试、分析与优化》。载于第*26*届*ACM*编程语言与操作系统架构支持国际会议论文集，ASPLOS '21，第559–572页，纽约，美国，2021年。计算机协会。[43] Martijn de Heus、Kyriakos Psarakis、Marios Fragkoulis 和 Asterios Katsifodimos。《无服务器有状态函数上的分布式事务》。载于第*15*届*ACM*分布式与事件系统国际会议，第31–42页，2021年。[44] Matei Zaharia、Mosharaf Chowdhury、Michael J. Franklin、Scott Shenker 和 Ion Stoica。《Spark：基于工作集的集群计算》。载于 Erich M. Nahum 和 Dongyan Xu 编，*HotCloud*。USENIX，2010年。[45] Shuhao Zhang、Juan Soto 和 Volker Markl。《事务流处理综述》。《*VLDB*期刊》，33(2)：451–479，2024年。[46] Kyriakos Psarakis、George Christodoulou、Marios Fragkoulis 和 Asterios Katsifodimos。《事务型云应用与（数据）流相得益彰》。载于第*15*届创新数据系统研究年会（*CIDR'25*）。*2025*年*1*月*19*日至*22*日，阿姆斯特丹，荷兰。，2025年。[47] Chaoyi Cheng、Mingzhe Han、Nuo Xu、Spyros Blanas、Michael D Bond 和 Yang Wang。《这是开发者还是数据库的责任？重新思考数据库中的并发控制》。载于第*13*届创新数据系统研究年会（*CIDR'23*）。*2023*年*1*月*8*日至*11*日，阿姆斯特丹，荷兰。，2023年。[48] Qian Li、Peter Kraft、Michael J. Cafarella、Çagatay Demiralp、Goetz Graefe、Christos Kozyrakis、Michael Stonebraker、Lalith Suresh 和 Matei Zaharia。《事务让调试变得轻松》。载于第*13*届创新数据系统研究会议，*CIDR*，2023年。[49] James C Corbett、Jeffrey Dean、Michael Epstein、Andrew Fikes、Christopher Frost、Jeffrey John Furman、Sanjay Ghemawat、Andrey Gubarev、Christopher Heiser、Peter Hochschild 等。《Spanner：谷歌的全球分布式数据库》。《*ACM*计算机系统汇刊（*TOCS*）》，31(3)：1–22，2013年。

[50] Rebecca Taft、Irfan Sharif、Andrei Matei、Nathan VanBenschoten、Jordan Lewis、Tobias Grieger、Kai Niemi、Andy Woods、Anne Birzin、Raphael Poss 等。《CockroachDB：弹性地理分布式 SQL 数据库》。载于 *2020*年*ACM SIGMOD* 数据管理国际会议论文集，第1493–1509页，2020年。[51] Philip A. Bernstein、Vassos Hadzilacos 和 Nathan Goodman。《数据库系统中的并发控制与恢复》。Addison-Wesley，1987年。[52] CAE 规范。《分布式事务处理：XA 规范》。X/Open，1991年。[53] Leslie Lamport。《Paxos 简明教程》。《*ACM SIGACT News*（分布式计算专栏）*32，4*（总第*121*期，*2001*年*12*月），第51–58页，2001年。[54] Harald Ng、Seif Haridi 和 Paris Carbone。《Omni-Paxos：打破部分连通性的壁垒》。载于第十八届欧洲计算机系统会议论文集，第314–330页，2023年。[55] Kyriakos Psarakis、Wouter Zorgdrager、Marios Fragkoulis、Guido Salvaneschi 和 Asterios Katsifodimos。《有状态实体：作为分布式数据流的面向对象云应用》。载于 *CIDR ’23*（摘要），*EDBT ’24*，2023年。[56] J. Gray 和 D.P. Siewiorek。《高可用计算机系统》。*Computer*，24(9)：39–48，1991年。[57] N. Malviya、A. Weisberg、S. Madden 和 M. Stonebraker。《重新思考主存联机事务处理恢复》。载于*2014*年*IEEE*第*30*届国际数据工程会议，第604–615页，2014年3月。[58] K Mani Chandy 和 Leslie Lamport。《分布式快照：确定分布式系统的全局状态》。载于*TOCS*，1985年。[59] Chandrasekaran Mohan、Don Haderle、Bruce Lindsay、Hamid Pirahesh 和 Peter Schwarz。《ARIES：一种支持细粒度锁定和部分回滚、采用预写式日志的事务恢复方法》。《*ACM*数据库系统汇刊（*TODS*）》，17(1)：94–162，1992年。[60] Diego Ongaro 和 John Ousterhout。《寻找一种易于理解的共识算法》。载于*ATC*，2014年。[61] Leslie Lamport。《兼职议会》。《*ACM*计算机系统汇刊》，1998年。[62] George Siachamis、Kyriakos Psarakis、Marios Fragkoulis、Arie van Deursen、Paris Carbone 和 Asterios Katsifodimos。《CheckMate：评估流式数据流的检查点协议》。载于*2024*年*IEEE*第*40*届国际数据工程会议（*ICDE*），第4030–4043页，2024年。[63] Tao Chen、Rami Bahsoon 和 Xin Yao。《自我感知与自适应云自动扩缩系统综述及分类》。《*ACM*计算调查》，51(3)，2018年6月。

[64] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[65] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 387–402, Boston, MA, July 2023. USENIX Association.

[66] George Siachamis、George Christodoulou、Kyriakos Psarakis、Marios Fragkoulis、Arie van Deursen 和 Asterios Katsifodimos。流处理自动伸缩器评估。载于第 *18* 届 *ACM* 分布式与事件驱动系统国际会议论文集，DEBS '24，第 110–122 页，美国纽约州纽约市，2024 年。计算机协会。[67] Rong Gu、Han Yin、Weichang Zhong、Chunfeng Yuan 和 Yihua Huang。Meces：通过优先级状态迁移实现有状态分布式流处理系统的低延迟伸缩。载于*2022* 年 *USENIX* 年度技术会议（*USENIX ATC 22*），第 539–556 页，2022 年。[68] Bonaventura Del Monte、Steffen Zeuch、Tilmann Rabl 和 Volker Markl。Rhino：高效管理流处理引擎中的超大规模分布式状态。载于*2020* 年 *ACM SIGMOD* 数据管理国际会议论文集，第 2471–2486 页，2020 年。[69] Moritz Hoffmann、Andrea Lattuada、Frank McSherry、Vasiliki Kalavri、John Liagouris 和 Timothy Roscoe。Megaphone：面向分布式流式数据流的延迟感知状态迁移。《*VLDB*基金会会刊》，12(9)：1002–1015，2019 年。[70] Aaron J. Elmore、Vaibhav Arora、Rebecca Taft、Andrew Pavlo、Divyakant Agrawal 和 Amr El Abbadi。Squall：面向分区式主内存数据库的细粒度实时重配置。载于*2015* 年 *ACM SIGMOD* 数据管理国际会议论文集，SIGMOD '15，第 299–313 页，美国纽约州纽约市，2015 年。计算机协会。[71] Mohammad Tari、Mostafa Ghobaei-Arani、Jafar Pouramini 和 Mohsen Ghorbian。无服务器计算中的自动伸缩机制：综合评述。计算机科学评论，53：100650，2024 年。[72] Ioana Baldini、Paul C. Castro、Kerry Shih-Ping Chang、Perry Cheng、Stephen J. Fink、Vatche Ishakian、Nick Mitchell、Vinod Muthusamy、Rodric M. Rabbah、Aleksander Slominski 和 Philippe Suter。无服务器计算：当前趋势与开放问题。载于*CoRR*，2017。[73] R. Arokia Paul Rajan。无服务器架构——云计算领域的一场革命。载于*2018* 年第十届高级计算国际会议（*ICoAC*），第 88–93 页，2018 年。

[74] Erwin van Eyk、Lucian Toader、Sacheendra Talluri、Laurens Versluis、Alexandru Uță 和 Alexandru Iosup。无服务器更胜一筹：从平台即服务迈向当今的云计算。*IEEEInternet Computing*，22(5)：8–17，2018 年。

[75] Joseph M. Hellerstein、Jose M. Faleiro、Joseph Gonzalez、Johann Schleier-Smith、Vikram Sreekanti、Alexey Tumanov 和 Chenggang Wu。无服务器计算：前进一步，后退两步。载于第九届创新数据系统研究双年会议，*CIDR 2019*，美国加州阿西洛马，*2019*年*1*月*13*日至*16*日，在线论文集。www.cidrdb.org，2019年。[76] Johann Schleier-Smith、Vikram Sreekanti、Anurag Khandelwal、Joao Carreira、Neer- aja J. Yadwadkar、Raluca Ada Popa、Joseph E. Gonzalez、Ion Stoica 和 David A. Patterson。无服务器计算的现状与未来：云计算的下一阶段。*Commun. ACM*，64(5)：76–84，2021年4月。[77] Samuel Kounev、Nikolas Herbst、Cristina L. Abad、Alexandru Iosup、Ian Foster、Prashant Shenoy、Omer Rana 和 Andrew A. Chien。无服务器计算：它是什么，又不是什么？*Commun. ACM*，66(9)：80–92，2023年8月。[78] Paris Carbone、Stephan Ewen、Gyula Fóra、Seif Haridi、Stefan Richter 和 Kostas Tzoumas。Apache Flink® 中的状态管理：一致的有状态分布式流处理。《*VLDB*基金会会刊》，10(12)：1718–1729，2017年。[79] Pedro Silvestre、Marios Fragkoulis、Diomidis Spinellis 和 Asterios Katsifodimos。Clonos：高可用流数据流的一致因果恢复。载于*SIGMOD*，2021年。[80] Alexander Thomson、Thaddeus Diamond、Shu-Chun Weng、Kun Ren、Philip Shao 和 Daniel J. Abadi。Calvin：面向分区数据库系统的快速分布式事务。载于*2012*年 *ACM SIGMOD* 国际数据管理会议论文集，SIGMOD '12，第1–12页，美国纽约州纽约市，2012年。计算机协会。[81] Yi Lu、Xiangyao Yu、Lei Cao 和 Samuel Madden。Aria：快速且实用的确定性联机事务处理数据库。载于*VLDB*，2020年。[82] Jon Gjengset、Malte Schwarzkopf、Jonathan Behrens、Lara Timbó Araújo、Martin Ek、Eddie Kohler、M Frans Kaashoek 和 Robert Morris。Noria：面向高性能Web应用的动态、部分有状态数据流。载于*OSDI*，2018年。[83] Vasiliki Kalavri、John Liagouris、Moritz Hoffmann、Desislava Dimitrova、Matthew Forshaw 和 Timothy Roscoe。三步足矣：面向分布式流数据流的快速、准确自动扩缩决策。载于第*13*届 {*USENIX*} 操作系统设计与实施研讨会（{*OSDI*} *18*），2018年。[84] Avrilia Floratou、Ashvin Agrawal、Bill Graham、Sriram Rao 和 Karthik Ramasamy。Dhalion：Heron 中的自调节流处理。《*VLDB*基金会会刊》，10(12)：1825–1836，2017年。

[85] Kyriakos Psarakis、Wouter Zorgdrager、Marios Fragkoulis、Guido Salvaneschi 和 Asterios Katsifodimos。有状态实体：面向对象的云应用作为分布式数据流。载于 第*13*届创新数据系统研究会议，*CIDR 2023*，荷兰阿姆斯特丹，*2023*年*1*月*8*日至*11*日。www.cidrdb.org，2023年。[86] Kyriakos Psarakis、Oto Mraz、George Christodoulou、George Siachamis、Marios Fragkoulis 和 Asterios Katsifodimos。Styx 实践：让事务型云应用变得简单。 *VLDB*基金会会刊，2025年。[87] Rodrigo Laigner、Kyriakos Christodoulou、George Psarakis、Asterios Katsifodimos 和 Yongluan Zhou。事务型云应用：现状、挑战与机遇。 *2025*年国际数据管理会议伴随会议，2025年。[88] Pat Helland。云中的可扩展联机事务处理：究竟有什么大不了？载于 第*14*届创新数据系统研究会议，*CIDR 2024*，美国夏威夷州查米纳德，*2024*年*1*月*14*日至*17*日。www.cidrdb.org，2024年。[89] Google Cloud Run 函数。https://cloud.google.com/functions，2024年。[90] Andreas Wittig 和 Michael Wittig。亚马逊云科技实践：*AWS* 深度指南。西蒙与舒斯特，2023年。[91] Hector Garcia-Molina 和 Kenneth Salem。Saga。*ACM SIGMOD* 记录，16(3)：249–259，1987年。[92] Qian Li、Peter Kraft、Kostis Kaffes、Athinagoras Skiadopoulos、Deeptaanshu Kumar、Jason Li、Michael J. Cafarella、Goetz Graefe、Jeremy Kepner、Christos Kozyrakis、Michael Stonebraker、Lalith Suresh 和 Matei Zaharia。DBOS 进展报告：面向数据库的操作系统。载于第*12*届创新数据系统研究会议，*CIDR 2022*，美国加州查米纳德，*2022*年*1*月*9*日至*12*日。www.cidrdb.org，2022年。[93] Alvin Cheung、Natacha Crooks、Joseph M Hellerstein 和 Mae Milano。云编程的新方向。载于第*11*届创新数据系统研究年会（*CIDR '21*），*2021*年*1*月*10*日至*13*日，美国查米纳德。，2021年。[94] Tianyu Li、Badrish Chandramouli、Sebastian Burckhardt 和 Samuel Madden。无服务器状态管理系统。载于*CIDR*，2024年。[95] Matei Zaharia、Tathagata Das、Haoyuan Li、Timothy Hunter、Scott Shenker 和 Ion Stoica。离散化流：大规模容错流式计算。载于*ACM SIGOPS* 第*24*届操作系统原理研讨会，*SOSP '13*，美国宾夕法尼亚州法明顿，*2013*年*11*月*3*日至*6*日，第423–438页，2013年。[96] Pat Helland。超越分布式事务：一位叛教者的观点。载于第三届创新数据系统研究双年会议，*CIDR 2007*，美国加州阿西洛马，*2007*年*1*月*7*日至*10*日，在线论文集，第132–141页。www.cidrdb.org，2007年。[97] Bas P Harenslak 和 Julian de Ruiter。使用 *Apache Airflow* 构建数据管道。西蒙与舒斯特

' 2021
年。

[98] Marco Serafini、Rebecca Taft、Aaron J Elmore、Andrew Pavlo、Ashraf Aboulnaga 和 Michael Stonebraker。《Clay：面向通用数据库模式的细粒度自适应分区》。《style id='2'>VLDB，2016年。[99] Marios Fragkoulis、Paris Carbone、Vasiliki Kalavri 和 Asterios Katsifodimos。《流处理系统演进综述》。《style id='5'>VLDB期刊，33(2)：507–541，2024年。[100] Jay Kreps、Neha Narkhede、Jun Rao 等。《Kafka：用于日志处理的分布式消息系统》。载于《style id='8'>NetDB论文集，第11卷，第1–7页，2011年。[101] Daniel J Abadi 和 Jose M Faleiro。《确定性数据库系统概述》。《style id='11'>ACM 通讯，61(9)：78–88，2018年。[102] Sebastian Burckhardt、Badrish Chandramouli、Chris Gillum、David Justo、Kon- stantinos Kallas、Connor McMahon、Christopher S Meiklejohn 和 Xiangfeng Zhu。《Netherite：高效执行无服务器工作流》。《style id='14'>VLDB，2022年。[103] Clifford Dale Krumvieda。《style id='17'>分布式机器学习：高效容错编程摘要。康奈尔大学，1993年。[104] Joe Armstrong。《style id='20'>Erlang编程：面向并发世界的软件。Pragmatic Bookshelf，2013年。[105] Derek Wyatt。《style id='23'>Akka并发。Artima Incorporation，2013年。[106] Eric Jonas、Johann Schleier-Smith、Vikram Sreekanti、Chia-Che Tsai、Anurag Khan- delwal、Qifan Pu、Vaishaal Shankar、Joao Carreira、Karl Krauth、Neeraja Yadwadkar 等。《简化云编程：伯克利视角下的无服务器计算》。载于*arXiv*，2019年。[107] Adil Akhter、Marios Fragkoulis 和 Asterios Katsifodimos。《有状态函数即服务实践》。载于*VLDB*，2019年。[108] Saulo S. de Toledo、Antonio Martini、Agata Przybyszewska 和 Dag I. K. Sjøberg。《微服务中的架构技术债务：一家大型公司的案例研究》。载于第二届技术债务国际会议论文集，TechDebt '19，第78–87页。IEEE出版社，2019年。[109] Tom Killalea。《微服务的隐性收益》。《style id='35'>ACM Queue，2016年。[110] Leslie Lamport、Robert Shostak 和 Marshall Pease。《拜占庭将军问题》。《style id='38'>ACM编程语言与系统汇刊，4(3)：382–401，1982年。[111] Paris Carbone、Marios Fragkoulis、Vasiliki Kalavri 和 Asterios Katsifodimos。《超越分析：流处理系统的演进》。载于*SIGMOD*，2020年。[112] Tyler Akidau、Alex Balikov、Kaya Bekiroğlu、Slava Chernyak、Josh Haberman、Reuven Lax、Sam McVeety、Daniel Mills、Paul Nordstrom 和 Sam Whittle。《Millwheel：互联网规模的容错流处理》。《style id='44'>VLDB基金会会刊，6(11)：1033–1044，2013年。

[113] Can Gencer、Marko Topolnik、Viliam Durina、Emin Demirci、Ensar B. Kahveci、Ali Gürbüz Ondrej Lukás、József Bartók、Grzegorz Gierlach、Frantisek Hartman、Ufuk Yilmaz、Mehmet Dogan、Mohamed Mandouh、Marios Fragkoulis 和 Asterios Katsifodimos。《Hazelcast Jet：第99.99百分位数下的低延迟流处理》。发表于*VLDB*，2021年。[114] Chenggang Wu、Vikram Sreekanti 和 Joseph M. Hellerstein。《面向无服务器计算的事务因果一致性》。发表于*SIGMOD*，2020年。[115] C. Wu、J. Faleiro、Y. Lin 和 J. Hellerstein。《Anna：适用于任意规模的键值存储》。发表于*ICDE*，2018年。[116] Jonathan Goldstein、Ahmed Abdelhamid、Mike Barnett、Sebastian Burckhardt、Badr- ish Chandramouli、Darren Gehring、Niel Lebeck、Christopher Meiklejohn、Umar Fa- rooq Minhas、Ryan Newton 等。《Ambrosia：为分布式应用提供高性能虚拟弹性》。发表于*VLDB*，2020年。[117] Raul Castro Fernandez、Matteo Migliavacca、Evangelia Kalyvianaki 和 Peter Pietzuch。《让命令式大数据处理中的状态显式化》。发表于*2014 USENIXannual technical conference (USENIX ATC 14)*，第49–60页，2014年。[118] A. Katsifodimos 和 M. Fragkoulis。《运维流处理：迈向可扩展且一致的事件驱动应用程序》。发表于*EDBT*，2019年。[119] Lorenzo Affetti、Alessandro Margara 和 Gianpaolo Cugola。《FlowDB：集成流处理与一致的状态管理》。发表于*DEBS*，2017年。[120] Doug Terry。《云数据库中的事务与可扩展性——我们能否兼得？》。发表于*USENIXAssociation*，2019年。[121] Robert Kallman、Hideaki Kimura、Jonathan Natkins、Andrew Pavlo、Alexander Rasin、Stanley Zdonik、Evan PC Jones、Samuel Madden、Michael Stonebraker、Yang Zhang 等。《H-Store：高性能分布式主存事务处理系统》。发表于*VLDB*，2008年。[122] Swaminathan Sivasubramanian。《Amazon DynamoDB：可无缝扩展的非关系型数据库服务》。发表于*SIGMOD*，2012年。[123] K. Mani Chandy、Jayadev Misra 和 Laura M. Haas。《分布式死锁检测》。发表于*ACMTrans.Comput.Syst.*，1983年。[124] Alexander Thomson 和 Daniel J. Abadi。《数据库系统采用确定性的理由》。*VLDB*基金会会刊，3(1–2):70–80，2010年9月。[125] Kun Ren、Alexander Thomson 和 Daniel J. Abadi。《确定性数据库系统优缺点的评估》。发表于*VLDB*，2014年。[126] Brian F Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。《使用YCSB对云服务系统进行基准测试》。发表于*Proceedings of the 1st ACMsymposium on Cloud computing*，第143–154页，2010年。

[127] Akon Dey、Alan Fekete、Raghunath Nambiar 和 Uwe Röhm。《YCSB+t：基准测试网络规模事务数据库》。发表于*2014*年*IEEE*第*30*届数据工程国际会议研讨会，第223–230页。IEEE，2014年。[128] Vikram Sreekanti、Chenggang Wu、Saurav Chhatrapati、Joseph E Gonzalez、Joseph M Hellerstein 和 Jose M Faleiro。《面向无服务器计算的容错垫片》。发表于*EuroSys*，2020年。[129] Srinath Setty、Chunzhi Su、Jacob R Lorch、Lidong Zhou、Hao Chen、Parveen Patel 和 Jinglei Ren。《利用意向锁实现云存储的容错承诺》。发表于*OSDI*，2016年。[130] Simon Shillaker 和 Peter Pietzuch。《FAASM：面向高效有状态无服务器计算的轻量级隔离》。发表于*2020USENIX Annual Technical Conference (USENIX ATC20)*，第419–433页，2020年。[131] Irina Botan、Peter M Fischer、Donald Kossmann 和 Nesime Tatbul。《事务流处理》。发表于*EDBT*，2012年。[132] Philipp Götze 和 Kai-Uwe Sattler。《事务流处理的快照隔离》。发表于*EDBT*，2019年。[133] Lorenzo Affetti、Alessandro Margara 和 Gianpaolo Cugola。《TSpoon：流处理器上的事务》。发表于《并行与分布式计算期刊》，2020年。[134] C Mohan、Bruce Lindsay 和 Ron Obermarck。《r*分布式数据库管理系统中的事务管理》。发表于*TODS*，1986年。[135] Xinan Yan、Linguan Yang、Hongbo Zhang、Xiayue Charles Lin、Bernard Wong、Kenneth Salem 和 Tim Brecht。《Carousel：面向全球分布式数据的低延迟事务处理》。发表于*SIGMOD*，2018年。[136] Kun Ren、Dennis Li 和 Daniel J Abadi。《SLOG：可串行化、低延迟的地理复制事务》。发表于*VLDB*，2019年。[137] Yu Gan、Yanqi Zhang、Dailun Cheng、Ankitha Shetty、Priyal Rathi、Nayan Katarki、Ariana Bruno、Justin Hu、Brian Ritchken、Brendon Jackson 等。《面向微服务的开源基准测试套件及其对云与边缘系统软硬件的影响》。发表于*Proceedingsof the Twenty-Fourth International Conferenceon Architectural Support forProgramming Languages and OperatingSystems*，第3–18页，2019年。[138] Jeyhun Karimov、Tilmann Rabl、Asterios Katsifodimos、Roman Samarev、Henri Heiskanen 和 Volker Markl。《分布式流数据处理系统的基准测试》。发表于*2018*年*IEEE*第*34*届数据工程国际会议，第1507–1518页。IEEE，2018年。[139] Francois Raab。《TPC-C——联机事务处理（OLTP）的标准基准测试》。载于 Jim Gray 编，《基准测试手册》。Morgan Kaufmann，1993年。

[140] Jeffrey Dean 和 Sanjay Ghemawat。《MapReduce：在大型集群上简化数据处理》。载于《ACM 通讯》，2008年。[141] Erik Meijer、Brian Beckman 和 Gavin Bierman。《LINQ：在 .NET 框架中协调对象、关系和 XML》。载于 SIGMOD，2006年。[142] Ankit Toshniwal、Siddarth Taneja、Amit Shukla、Karthik Ramasamy、Jignesh M Patel、Sanjeev Kulkarni、Jason Jackson、Krishna Gade、Maosong Fu、Jake Donham 等。《Twitter Storm》。载于 SIGMOD，2014年。[143] Derek G Murray、Frank McSherry、Rebecca Isaacs、Michael Isard、Paul Barham 和 Martín Abadi。《Naiad：及时的数据流系统》。载于 ACM SOSP，2013年。[144] Andrea Lattuada、Frank McSherry 和 Zaheer Chothia。《Faucet：数据流引擎中一种用户级模块化流量控制技术》。载于 ACM SIGMOD BeyondMR 研讨会。ACM，2016年。[145] Apache Spark 项目。http://spark.apache.org/。[146] Paul Hudak、Simon Peyton Jones、Philip Wadler、Brian Boutel、Jon Fairbairn、Joseph Fasel、María M Guzmán、Kevin Hammond、John Hughes、Thomas Johnsson 等。《Haskell 编程语言报告》。《SIGPLAN 通讯》，1992年。[147] Robert Harper、David MacQueen 和 Robin Milner。《Standard ML》。计算机科学系，爱丁堡大学，1986年。[148] John C. Reynolds。《续延的发现》。《LISP 与符号计算》，1993年。[149] Gabriela Jacques-Silva、Fang Zheng、Daniel Debrunner、Kun-Lung Wu、Victor Dogaru、Eric Johnson、Michael S picer 和 Ahmet Erdem Sariyüce。《一致性区域：保证 IBM Streams 中的元组处理》。《VLDB基金会会刊》，9(13):1341–1352，2016年9月。[150] Henry G. Baker。《Cons 不应复制其参数，第二部分：Cheney 与 M.T.A.》。《SIGPLAN 通讯》，1995年。[151] Gérard Berry 和 Georges Gonthier。《Esterel 同步编程语言：设计、语义与实施》。《计算机编程科学》，19(2):87–152，1992年。[152] William W. Wadge 和 Edward A. Ashcroft。《LUCID：数据流编程语言》。Academic Press Professional, Inc.，美国，1985年。[153] Michael Stonebraker 和 Ariel Weisberg。《VoltDB 主存数据库管理系统》。《IEEE 数据工程通报》，36(2):21–27，2013年。[154] Stephanie Wang、John Liagouris、Robert Nishihara、Philipp Moritz、Ujval Misra、Alexey Tumanov 和 Ion Stoica。《谱系暂存：将容错移出关键路径》。载于 ACM SOSP，2019年。

[155] Jim Verheijde、Vassilios Karakoidas、Marios Fragkoulis 和 Asterios Katsifodimos。《S-query：揭开流处理器内部状态的黑盒》。载于《2022年 IEEE 第38届国际数据工程会议（ICDE）》，第1314–1327页。IEEE，2022年。[156] Philip A Bernstein、Mohammad Dashti、Tim Kiefer 和 David Maier。《面向参与者的数据库中的索引》。载于 CIDR，2017年。[157] Peter Bailis、Alan Fekete、Ali Ghodsi、Joseph M Hellerstein 和 Ion Stoica。《利用 RAMP 事务实现可扩展的原子可见性》。《ACM 数据库系统汇刊（TODS）》，41(3):1–45，2016年。[158] L Peter Deutsch 和 Allan M Schiffman。《Smalltalk-80 系统的高效实施》。载于 SIGACT-SIGPLAN，1984年。[159] Tamer Eldeeb 和 Philip A Bernstein。《云中分布式参与者的事务》。微软研究院，2016年。[160] K Venkatesh Emani、Tejas Deshpande、Karthik Ramachandra 和 S Sudarshan。《Dbridge：将命令式代码转换为 SQL》。载于 SIGMOD，2017 年。[161] Gagan Gupta 和 Gurindar S Sohi。《多核架构上顺序命令式程序的数据流执行》。载于 MICRO，2011年。[162] 微软公司。《Azure Logic Apps》。https://azure.microsoft.com/en-us/ products/logic-apps/，2025年。访问日期：2025年1月14日。[163] Zhipeng Jia 和 Emmett Witchel。《Nightcore：面向延迟敏感型交互式微服务的高效、可扩展无服务器计算》。载于《第26届 ACM 编程语言与操作系统架构支持国际会议论文集》，第152–166页，2021年。[164] Jonas Spenger、Paris Carbone 和 Philipp Haller。《Portals：面向有状态无服务器的扩展数据流流处理》。载于《2022年 ACM SIGPLAN 编程与软件新思想、新范式与反思国际研讨会论文集》，第153–171页，2022年。[165] Shadi A. Noghabi、Kartik Paramasivam、Yi Pan、Navina Ramesh、Jon Bringhurst、Indranil Gupta 和 Roy H. Campbell。《Samza：领英上的有状态可扩展流处理》。《VLDB基金会会刊》，10，2017年。[166] Michael Stonebraker、U˘gur Çetintemel 和 Stan Zdonik。《实时流处理的8项要求》。《ACM SIGMOD 记录》，34(4):42–47，2005年。[167] Xinjing Zhou、Xiangyao Yu、Goetz Graefe 和 Michael Stonebraker。《Lotus：在单线程分区数据库上实现可扩展的多分区事务》。《VLDB基金会会刊》，15(11):2939–2952，2022年。[168] Yanhua Mao、Flavio P. Junqueira 和 Keith Marzullo。《Mencius：构建高效的广域网复制状态机》。载于《第8届 USENIX 操作系统设计与实现研讨会（OSDI 08）》，2008年。

[169] Stephanie Wang、Eric Liang、Edward Oakes、Ben Hindman、Frank Sifei Luan、Audrey Cheng 和 Ion Stoica。Ownership：面向细粒度任务的分布式 future 系统。发表于 第 *18* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 21*），第 671–686 页。USENIX 协会，2021年4月。[170] Sidi Mohamed Beillahi、Ahmed Bouajjani、Constantin Enea 和 Shuvendu Lahiri。异步化的自动综合。发表于 国际静态分析研讨会，第 135–159 页。Springer，2022年。[171] Tianyu Li、Badrish Chandramouli、Sebastian Burckhardt 和 Samuel Madden。Darq matter binds everything：通过弹性步骤实现高性能、可组合的云编程。《*ACM* 数据管理会议论文集》，1(2)，2023年6月。[172] E. N. (Mootaz) Elnozahy、Lorenzo Alvisi、Yi-Min Wang 和 David B. Johnson。消息传递系统中回滚恢复协议综述。《*ACM*计算调查》，34(3):34，2002年。[173] Temporal。推出 Temporal .NET——在 .NET 中编写确定性工作流。https://temporal.io/blog/introducing-temporal-dotnet，2025年。访问日期：2025年1月14日。[174] IETF。幂等键 HTTP 标头字段。
h
t
t
p
s://
www.ietf.org/archive/id/draft-ietf-httpapi-idempotency-key-header-05.txt，2025年。访问日期：2025年1月14日。[175] Sheng Qi、Xuanzhe Liu 和 Xin Jin。Halfmoon：对数最优的容错有状态无服务器计算。发表于第 *29* 届操作系统原理研讨会论文集，第 314–330 页，2023年。[176] Minio。https://min.io/。[177] David H. Liu、Amit Levy、Shadi A. Noghabi 和 Sebastian Burckhardt。以少胜多：无需编排器即可编排无服务器应用。发表于第 *20* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 23*），第 1505–1519 页，2023年。[178] Derek G Murray、Malte Schwarzkopf、Christopher Smowton、Steven Smith、Anil Madhavapeddy 和 Steven Hand。 {CIEL}：用于分布式 {数据流} 计算的通用执行引擎。发表于第 *8* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 11*），2011年。[179] Zihao Zhang、Huiqi Hu、Xuan Zhou 和 Jiang Wang。Starry：基于半领导者架构的多主事务处理。《*VLDB*基金会会刊》，16(1):77–89，2022年。[180] Michael Abebe、Brad Glasbergen 和 Khuzaima Daudjee。Morphosys：分布式数据库系统的自动物理设计变形。《*VLDB*基金会会刊》，13(13):3573–3587，2020年。

[181] Yann Collet 和 Murray S. Kucherawy。Zstandard 压缩及“application/zstd” 媒体类型。 *RFC*，8878:1–45，2021年。[182] Bonaventura Del Monte、Steffen Zeuch、Tilmann Rabl 和 Volker Markl。Rhino：高效管理流处理引擎中的超大规模分布式状态。发表于 *ACMSIGMOD*，2020年。[183] Sudipto Das、Shoji Nishimura、Divyakant Agrawal 和 Amr El Abbadi。Albatross：利用实时数据迁移，为云端共享存储数据库实现轻量级弹性。 *VLDB*基金会会刊，4(8):494–505，2011年。[184] To m Z. J. Fu、Jianbing Ding、Richard T. B. Ma、Marianne Winslett、Yin Yang 和 Zhenjie Zhang。DRS：面向实时流分析的自动扩缩容。 *IEEE/ACM* 网络汇刊，25(6):3338–3352，2017年。[185] Kubernetes 水平 Pod 自动扩缩。 h ttps://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/。访问日期：2025-6-16。[186] Ioana Baldini、Paul C. Castro、Kerry Shih-Ping Chang、Perry Cheng、Stephen J. Fink、Vatche Ishakian、Nick Mitchell、Vinod Muthusamy、Rodric M. Rabbah、Aleksander Slominski 和 Philippe Suter。无服务器计算：当前趋势与开放问题。*CoRR*，2017年。

图目录

1.1 云计算抽象的发展演变及所需的人工参与.31.2 结账场景中两阶段提交与 Saga 事务协议的比较，该场景跨越三个微服务（订单、库存和支付）。状态变更阶段以红色标记。本例中，两阶段提交需要四个时间槽才能提交，而 Saga 只需要两个。.71.3 在典型场景中，操作人员根据预期峰值配置机器，导致大量资源浪费。使用无服务器解决方案时，操作人员无需配置任何机器，资源配置与实际负荷相匹配，从而减少浪费。.101.4 本论文的可视化大纲，展示了各章节及其中涵盖的主要思想。152.1 在单体应用中，开发者专注于应用逻辑，而事务数据库负责状态管理和故障恢复。在分布式云应用中，开发面临更多挑战（例如故障、恰好一次消息传递，以及为实现原子性和可扩展性而进行的编排）。理想的运行时应在提升可扩展性的同时，提供与单体应用相同的状态一致性和编程便捷性，且无需开发者参与。.192.2 将单体应用转换为可扩展、低延迟服务架构的三步流程。.203.1 Flink StateFun 集群架构.313.2 Flink StateFun 的原始通信流程.323.3 T-Statefun 中事务的通信流程。.373.4 StateFun 基准测试设计.413.5 原始 StateFun 与带协调器函数的 StateFun 的最大吞吐量。.423.6 对比原始 StateFun（OS）与带协调器函数的 StateFun（CF）在不同吞吐量下，针对只读和只写工作负载的延迟图。.433.7 工作负载中转账操作比例不断增加时的最大吞吐量.443.8 对包含 100 个和 5000 个键、转账比例各不相同且不发生回滚的工作负载，详细展示其锁定行为。箱线图显示第 5 和第 95 百分位数。.45

3.9 对于转账操作占比为 50% 和 100% 的工作负载，已回滚转账操作占不同比例时的吞吐量.45 3.10 对于包含 5000 个键的系统，在转账操作占比不同的工作负载下，StateFun 工作节点数量不同时的最大吞吐量.46 3.11 在各自最大吞吐量的 80% 负载下，对比 Saga 与两阶段提交协调器函数在不同键数量和事务比例下的延迟.47 3.12 对比 CockroachDB 和 Flink StateFun 在转账操作占比不同的工作负载下的最大吞吐量（其余操作以相同概率为读取或更新操作）。两个系统均部署了 3 个实例，每个实例配备 4 个 CPU.48 4.1 两个有状态实体：User 和商品。命令式函数的内容被拆分为多个函数，这些函数访问给定实体的共享状态。随后，这些函数被编码为有状态数据流，可在分布式流数据流引擎中执行。因此， *i*)命令式代码以基于事件的方式执行，无需阻塞； *ii*) 代码仍保留精确一次处理保证，程序员无需编写状态管理、调用重试或幂等性等故障处理代码.57 4.2 包含五个实体的逻辑数据流图，重点展示图 4.1 中的 User 实体.58 4.3 YCSB（每秒 100 个请求）中齐夫分布和均匀键分布下的平均延迟（第 99 百分位数）.63 4.4 M 工作负载在输入吞吐量不断增加时的平均延迟和第 99 百分位延迟.64 5.1 对于事务型工作负载（第 5.7 节），Styx 的性能至少比最先进水平高一个数量级。图中显示中位数（柱形）/99百分位（误差线）延迟和吞吐量。对于延迟图，输入吞吐量为每秒 2000 个事务（TPS）；对于吞吐量图，我们报告系统在亚秒级延迟下达到的吞吐量。. . . . . 68 5.2 Styx 中 Deathstar 的酒店/航班预订。从第 9 至 14 行开始，异步调用*reserve_hotel* 和 *reserve_flight* 函数。最后，在第 16 至 17 行，存储预订信息。在 Styx 中，事务逻辑和容错逻辑均由系统内部处理.72 5.3 Styx 中的有状态函数执行。在每个工作节点中，一个协程负责对传入事务进行排序，另一个协程负责处理这些事务。本例中，事务（创建_预订）由两个函数组成：预留_酒店和预留_航班。函数可以访问本地状态（预留_酒店），也可以对不同分区执行远程调用（预留_航班）。该远程调用使用分区器来定位存储该分区的正确工作进程.74

5.4 Styx 中的事务执行流水线分为 4 个部分。首先，每个外部请求（R_i）都会被排序为一个事务，并分配一个唯一标识符。随后，各事务执行其应用逻辑、访问本地键并发起远程函数调用。事务执行期间，Styx 会跟踪其访问的键（$[R/W]_i$），并逐步构建调用图。接着，Styx 会提交那些不涉及未解决冲突的事务，无需执行锁定。例如，我们观察到，工作节点 W_1 和 W_2 能够在 T_1 与 T_2 访问相同键的同时提交 $C_1 = C_2 = \{T_1\}$；尽管它的 ID 最小。最后，我们通过基于锁的机制解决冲突，并提交所有事务（$C_2 = \{T_2, T_3\}$，$C_3 = \{T_3\}$）.75 5.5 Styx 中使用三个排序器进行 TID 分配的示例。它们的标识符$\{1,2,3\}$按照公式（5.1）生成以下序列： $S_1 = \{1,4,...,k\}$、 $S_2 = \{2,5,...,m\}$、$S_3 = \{3,6,...,n\}$。. . . . . 76 5.6 异步函数调用链。给定的根函数调用在执行过程中可以调用其他函数。随着函数执行的推进，原始确认$(3/3)$会拆分为多个部分，每个函数都会获得自己的确认份额。例如，在此函数执行过程中，根函数调用了另外三个函数，因此确认份额被均分为三份。后续调用也是如此，调用方会进一步拆分其确认份额。末端（蓝色）调用（即不再发起其他调用的函数调用）的确认份额总和恰好为 $3/3$，因此根函数可以报告执行完成。. . . 78 5.7 如果不使用函数缓存（左），事务执行将沿着较深的调用图进行；消息将按顺序发送，数量等于函数调用次数（5）加上确认数（2）。Styx 的函数缓存优化（右）会在 t_0 和 t_1之间的基于锁的提交阶段实现并发函数执行，并仅异步发送五条确认.79 5.8 Styx 中使用增量映射的增量快照.83 5.9 外部系统调用关键点与 Styx.85 5.10 不同场景下的评估。T-Statefun 不支持 Deathstar 工作负载所需的范围查询。只有 Styx 支持 TPC-C。. . . 87 5.11 使用 YCSB-T（倾斜分布）评估不同争用程度（0.0 - 0.999）下的延迟。我们以两种不同的输入吞吐量运行 Styx，以清晰展示其在争用下的行为。请注意，Styx 和 T-Statefun 会执行所有事务直至完成（中止率=0）.88 5.12 Styx 在 YCSB-T 上的可扩展性，涉及不同占比的多分区事务. 89 5.13 Styx 的快照对性能的影响.90 5.14 Styx 在恢复期间的行为.90 5.15 使用 ~20 GB TPC-C 状态时，Styx 增量快照的行为。. . . . 91 5.16 微服务范式（图 5.16a ）与 Styx （图 5.16b ）的比较 .93

5.17 Styx 监控仪表板。 .9 4

6.1 纪元阶段与在线状态迁移的对齐。在阶段 中，事务决定其调度图，因此相关键需要立即迁移。相反，在 和 中，键可以异步迁移，而不会干扰事务协议。 . . . . 1016.2 迁移请求前后的状态分布，该请求移除了 $worker_4$。迁移后，我们看到一个事务（$T_1 = \{k_1, k_5\}$）的示例。 T_1与键 k_1 和 k_5交互，这意味着虽然 k_1 已就位，k_5 仍需按需迁移，正如图 6.1 所示，迁移阶段 所指出的那样。其余键在没有其他事务与之交互的情况下，可以在迁移阶段 异步迁移。 . 1026.3 大状态（10GB）缩容时停止并重启（S&R）与在线迁移在 TPC-C 和 YCSB 中的表现：延迟（上排）和吞吐量（下排）。 .1046.4 小状态 1GB 缩容：延迟（上排）和吞吐量（下排）。 .1046.5 大状态 10 GB 扩容：延迟（上排）和吞吐量（下排）。 1056.6 小状态 1GB 扩容：延迟（上排）和吞吐量（下排）。 105

表格列表

3.1 协调器函数的 Python API。 .343.2 StateFun 与 CockroachDB 的延迟比较：每个系统均以图 3.12 所示测得的最大吞吐量的 80% 运行 . . . . 493.3 Beldi 与 StateFun 的延迟比较（*实验受到节流，运行吞吐量较低 ≈ 4K，即实验持续时间更长）. 505.1 工作负载特征。 .865.2 使用 YCSB-T 评估不同争用程度下的 Boki 和 Beldi。我们报告中止率和已提交事务率，并省略延迟，因为这些系统并未将所有事务执行至完成。两者均以最大可持续吞吐量运行。 .885.3 所有系统的性能细分。（中位延迟——占总量的百分比） .89

致谢

Doing a PhD can often feel like a solitary endeavor; however, this journey would not have been possible without the support of many people, for which I am deeply grateful.

Asterios. Thank you for all the "rope" you gave me to pursue the things I truly enjoyed during this PhD, as well as for the opportunities and life advice along the way.

Geert-Jan. Thank you for our philosophical discussions and for your support during some of the most difficult moments of my PhD.

Thesis Committee. Thank you for the time you dedicated to evaluating this thesis and for honoring me with your participation in the defense. Special thanks to Paris for always answering my questions about his own PhD and distributed systems in general.

George C. You may have joined during my final years, but your critical assistance and fresh perspective made a profound difference. I can say with certainty that I would not be finishing this PhD without you.

Marios. Thank you for your steady guidance throughout these years. Your clarity, patience, and calm perspective helped me navigate difficult moments and grow as a researcher and as a person.

George S. & Christos. We started as academic brothers, but I now consider you close to real brothers. Thank you for all the great moments, your tolerance, and your help.

My Master Students. I hope I have supported your learning journey as much as you have enriched mine.

WIS Group. Thank you to all colleagues in the WIS group for the supportive and stimulating environment, and for making daily life at the university genuinely enjoyable.

To everyone who opened this document to check if their name appears here, thank you. To my friends, both near and far, and to my family: your presence, messages, visits, and timely distractions kept me grounded and sane throughout this journey. Nothing I have achieved would have been possible without your love, patience, and faith in me.

Kyriakos
Delft, January 2026

个人简历

Kyriakos Psarakis

08-12-1994	Born in Chania, Greece

工作经历

2025-present	Software Engineer, Ververica GmbH, Remote
2023	Research Intern, Huawei Technologies R&D, The UK
2019-2020	Research Intern, ING Group, The Netherlands

教育经历

2021-2025	Doctor of Philosophy (Ph.D.), Computer Science Delft University of Technology, The Netherlands
2018-2020	Master of Science (M.Sc.), Computer Science Delft University of Technology, The Netherlands
2012-2018	Diploma (M.Eng.), Electrical and Computer Engineering Technical University of Crete, Greece

出版物列表

1. J. Arns、H. Ng、**K. Psarakis**、A. Katsifodimos 和 P. Carbone。《Event Horizon：快速地理分布式操作的非对称依赖》，发表于创新数据系统研究会议（CIDR），2026。2. **K. Psarakis**、G. Christodoulou、G. Siachamis、M. Fragkoulis 和 A. Katsifodimos。《Styx 中的状态迁移：迈向无服务器事务函数》，审稿中，2025。3. **K. Psarakis**、O. Mraz、G. Christodoulou、G. Siachamis、M. Fragkoulis 和 A. Katsifodimos。《Styx 实战：轻松构建事务型云应用》，发表于超大型数据库会议（VLDB），2025。4. **K. Psarakis**、G. Christodoulou、G. Siachamis、M. Fragkoulis 和 A. Katsifodimos。《Styx：流式数据流上的事务型有状态函数》，发表于 ACM 数据管理特别兴趣小组会议（SIGMOD），2025。5. R. Laigner、G. Christodoulou、**K. Psarakis**、A. Katsifodimos 和 Y. Zhou。《事务型云应用：现状、挑战与机遇》，发表于 ACM 数据管理特别兴趣小组会议（SIGMOD），2025。6. **K. Psarakis**、G. Christodoulou、M. Fragkoulis 和 A. Katsifodimos。《事务型云应用随数据流而行》，发表于创新数据系统研究会议（CIDR），2025。7. **K. Psarakis**、W. Zorgdrager、M. Fragkoulis、G. Salvaneschi 和 A. Katsifodimos。《有状态实体：作为分布式数据流的面向对象云应用》，发表于国际数据库技术扩展会议（EDBT），2024。8. G. Siachamis、**K. Psarakis**、M. Fragkoulis、A. van Deursen、P. Carbone 和 A. Katsifodimos。《CheckMate：流式数据流检查点协议评估》，发表于 IEEE 国际数据工程会议（ICDE），2024。9. G. Siachamis、G. Christodoulou、**K. Psarakis**、M. Fragkoulis、A. van Deursen、A. Katsifodimos。《流处理自动伸缩器评估》，发表于 ACM 分布式与事件驱动系统会议（DEBS），2024。10. G. Siachamis、**K. Psarakis**、M. Fragkoulis、O. Papapetrou、A. van Deursen、A. Katsifodimos。《自适应分布式流相似连接》，发表于 ACM 分布式与事件驱动系统会议（DEBS），2023。11. A. Ionescu、K. Patroumpas、**K. Psarakis**、G. Chatzigeorgakidis、D. Collarana、K. Barenscher、D. Skoutas、A. Katsifodimos 和 S. Athanasiou。《Topio：用于地理空间数据交易的开源网络平台》，发表于国际网络工程会议（ICWE），2023。3 12. A. Ionescu、A. Alexandridou、L. Ikonomou、**K. Psarakis**、K. Patroumpas、G. Chatzigeorgakidis、D. Skoutas、S. Athanasiou、R. Hai 和 A. Katsifodimos。《Topio市场：地理空间数据的搜索与发现》，发表于国际数据库技术扩展会议（EDBT），2023。

13. **K. Psarakis**、W. Zorgdrager、M. Fragkoulis、G. Salvaneschi 和 A. Katsifodimos。《有状态实体：作为分布式数据流的面向对象云应用》，发表于创新数据系统研究会议（CIDR），2023。14. M. de Heus、**K. Psarakis**、M. Fragkoulis 和 A. Katsifodimos。《利用有状态数据流实现跨无服务器函数的事务》，发表于 Elsevier Information Systems，2022。15. C. Koutras、**K. Psarakis**、G. Siachamis、A. Ionescu、M. Fragkoulis、A. Bonifati 和 A. Katsifodimos。《Valentine 实战：大规模匹配表格数据》，发表于超大型数据库会议（VLDB），2021。3 16. M. de Heus、**K.Psarakis**、M. Fragkoulis 和 A. Katsifodimos。《无服务器有状态函数上的分布式事务》，发表于 ACM 分布式与事件驱动系统会议（DEBS），2021。17. C. Koutras、G. Siachamis、A. Ionescu、**K. Psarakis**、J. Brons、M. Fragkoulis、C. Lofi、A. Bonifati 和 A. Katsifodimos。《Valentine：评估用于数据集发现的匹配技术》，发表于 IEEE 国际数据工程会议（ICDE），2021。

已收录于本论文。曾获最佳论文或演示奖。

SIKS 学位论文丛书

2016 01 Syed Saiden Abbas (RUN)，人类与机器的形状识别 02 Michiel Christiaan Meulendijk (UU)，通过决策支持优化药物审查：开出更容易接受的药 03 Maya Sappelli (RUN)，语境中的知识工作：以用户为中心的知识工作者支持 04 Laurens Rietveld (VUA)，关联数据的发布与使用 05 Evgeny Sherkhonov (UvA)，扩展无环查询：包含及其在解释缺失答案中的应用 06 Michel Wilson (TUD)，不确定环境下的鲁棒调度 07 Jeroen de Man (VUA)，衡量并建模虚拟培训中的负面情绪 08 Matje van de Camp (TiU)，通往过去的链接：从非结构化数据构建历史社交网络 09 Archana Nottamkandath(VUA)，如何信任有关文化遗物的众包信息 10 George Karafotias (VUA)，进化算法的参数控制 11 Anne Schuth (UvA)，从用户身上学习的搜索引擎 12 Max Knobbout (UU)，用于建模和验证规范多智能体系统的逻辑 13 Nana Baah Gyan (VUA)，万维网、语音技术与西非农村发展——一种信息通信技术促进发展方法 14 Ravi Khadka (UU)，重新审视遗留软件系统现代化 15 Steffen Michels (RUN)，混合概率逻辑——理论方面、算法与实验 16 Guangliang Li (UvA)，能够从人类奖励中学习的社会智能自主代理 17 Berend Weel (VUA)，迈向机器人有机体的具身进化 18 Albert Meroño Peñuela (VUA)，完善万维网上的统计数据 19 Julia Efremova (TU/e)，从族谱数据中挖掘社会结构 20 Daan Odijk (UvA)，新闻与网络搜索中的语境和语义 21 Alejandro Moreno Célleri (UT)，从传统游戏空间到互动游戏空间：对互动捉人游戏场中玩家行为的自动分析 22 Grace Lewis (VUA)，面向网络边缘计算系统的软件架构策略 23 Fei Cai (UvA)，信息检索中的查询自动补全 24 Brend Wanders (UT)，数据的再利用与概率集成：一种迭代的、独立于数据模型的方法

25 Julia Kiseleva（TU/e），利用情境信息理解搜索与浏览行为 26 Dilhan Thilakarathne（VUA），失控还是掌控：探索用于研究人类意识与控制在行为选择中作用的计算模型，并应用于航空与能源管理领域 27 Wen Li（TUD），理解社交媒体上的地理空间信息 28 Mingxin Zhang（TUD），大规模基于智能体的社会模拟——疫情预测与控制研究 29 Nicolas Höning（TUD），去中心化电力系统中的峰值削减——灵活规划的市场与价格 30 Ruud Mattheij（TiU），眼见为实 31 Mohammad Khelghati（UT），深网内容监测 32 Eelco Vriezekolk（UT），评估危机组织的电信服务可用性风险 33 Peter Bloem（UvA），单样本统计：从一个示例中学习的练习 34 Dennis Schunselaar（TU/e），可配置流程树：引出、分析与执行 35 Zhaochun Ren（UvA），社交媒体监控：摘要、分类与推荐 36 Daphne Karreman（UT），超越R2D2：针对机器人特定形态优化的非语言交互行为设计 37 Giovanni Sileno（UvA），协调法律与行动——概念与计算探究 38 Andrea Minuto（UT），重要的材料——智能材料与艺术及交互设计相遇 39 Merijn Bruijnes（UT），可信的嫌疑人智能体；人工嫌疑人的响应与人际风格选择 40 Christian Detweiler（TUD），将价值观纳入设计 41 Thomas King（TUD），治理治理：用于分析制度设计与执行治理的形式化框架 42 Spyros Martzoukos（UvA），双语对齐语料库的组合与构成方面 43 Saskia Koldijk（RUN），情境感知的压力自我管理支持：从理论到实践 44 Thibault Sellam（UvA），用于数据库探索的自动化助手 45 Bram van de Laar（UT），体验脑机接口控制 46 Jorge Gallego Perez（UT），让你快乐的机器人 47 Christina Weber（UL），实时前瞻——动态创新网络的准备工作 48 Tanja Buttler（TUD），收集经验教训 49 Gleb Polevoy（TUD），项目中的参与与互动：博弈论分析 50 Yan Wang（TiU），梦想之桥：迈向信息技术赋能的服务供应链运营绩效协调方法

2017 01 Jan-Jaap Oerlemans（UL），研究网络犯罪 02 Sjoerd Timmer（UU），运用论证设计和理解法证贝叶斯网络 03 Daniël Harold Telgen（UU），网格制造；采用信息物理方法的自主产品与可重构制造机器 04 Mrunal Gawade（CWI），列存储中的多核并行 05 Mahdieh Shadi（UvA），协作行为 06 Damir Vandic（EUR），用于网络商品搜索的智能信息系统 07 Roel Bertens（UU），信息洞察：从抽象到异常 08 Rob Konijn（VUA），检测有趣的差异：利用健康保险数据进行数据挖掘、离群点检测与子群发现 09 Dong Nguyen（UT），作为社会与文化数据的文本：关于文本变异的计算视角 10 Robby van Delden（UT），（引导）交互式游戏行为 11 Florian Kunneman（RUN），Twitter中时间与情感模式的建模 # anticipointment 12 Sander Leemans（TU/e），具有保证的稳健流程挖掘 13 Gijs Huisman（UT），社交触觉技术——通过触觉技术拓展社交触觉的触达范围 14 Shoshannah Tekofsky（TiU），你所玩的游戏塑造了你：从电子游戏行为中建模玩家特征 15 Peter Berck（RUN），基于记忆的文本校正 16 Aleksandr Chuklin（UvA），理解与建模现代搜索引擎的用户 17 Daniel Dimov（UL），众包在线争议解决 18 Ridho Reinanda（UvA），用于搜索的实体关联 19 Jeroen Vuurens（UT），信息检索中术语、文本与语义向量的邻近性 20 Mohammadbashir Sedighi（TUD），促进知识共享参与度：感知收益、成本与可见性的作用 21 Jeroen Linssen（UT），元数据在交互式叙事与严肃游戏中的重要性（世界游戏） 22 Sara Magliacane（VUA），不确定性下因果推断的逻辑 23 David Graus（UvA），感兴趣实体——数字痕迹中的发现 24 Chang Wang（TUD），利用可供性实现高效的机器人学习 25 Veruska Zamborlini（VUA），临床指南的知识表示及其在多病共存分析与文献检索中的应用 26 Merel Jung（UT），能够理解并回应人类触觉的社交智能机器人 27 Michiel Joosse（UT），研究社交机器人的定位与注视行为：人们的偏好、感知与行为 28 John Klein（VUA），复杂情境下的架构实践 29 Adel Alhuraibi（TiU），从IT—业务战略协调到绩效：社会创新与企业IT治理的调节中介模型

30 Wilma Latuny (TiU)，面部表情的力量 31 Ben Ruijl (UL)，量子场论计算计算方法的进展 32 Thaer Samar (RUN)，网络档案中内容的访问与检索 33 Brigit van Loggem (OU)，迈向软件文档的设计原理：计算机媒介活动模型 34 Maren Scheffel (OU)，学习分析评估框架 35 Martine de Vos (VUA)，解读自然科学电子表格 36 Yuanhao Guo (UL)，基于高通量成像的表型表征形状分析 37 Alejandro Montes Garcia (TU/e)，WiBAF：支持用户掌控隐私的浏览器内自适应框架 38 Alex Kayal (TUD)，规范性社交应用 39 Sara Ahmadi (RUN)，利用人类听觉系统特性和压缩感知方法提升自动语音识别的抗噪性 40 Altaf Hussain Abro (VUA)，引导你的心智：计算探索情绪、欲望与社会支持相关的人类控制，以应用于以人为本的支持系统 41 Adnan Manzoor (VUA)，关注健康生活方式：探索心理过程与智能环境，为健康生活方式提供支持 42 Elena Sokolova (RUN)，利用混合与缺失数据进行因果发现及其在注意缺陷多动障碍数据集中的应用 43 Maaike de Boer (RUN)，视频检索中的语义映射 44 Garm Lucassen (UU)，理解用户故事——敏捷需求工程中的计算语言学 45 Bas Testerink (UU)，去中心化运行时规范执行 46 Jan Schneider (OU)，基于传感器的学习支持 47 Jie Yang (TUD)，加速群体知识创建 48 Angel Suarez (OU)，协作式探究学习 2018 01 Han van der Aa (VUA)，流程表示的比较与对齐 02 Felix Mannhardt (TU/e)，多视角流程挖掘 03 Steven Bosems (UT)，幸福感因果模型：知识建模、情境感知应用的模型驱动开发与行为预测 04 Jordan Janeiro (TUD)，面向数据中心型工程任务中诊断团队的灵活协调支持 05 Hugo Huurdeman (UvA)，支持信息搜寻过程的复杂动态 06 Dan Ionita (UT)，社会技术系统的模型驱动信息安全风险评估 07 Jieting Luo (UU)，多智能体系统中机会主义的形式化描述 08 Rick Smetsers (RUN)，软件系统模型学习的进展 09 Xu Xie (TUD)，离散事件仿真中的数据同化 10 Julienka Mollee (VUA)，迈向未来：利用智能技术支持身体活动行为改变

11 Mahdi Sargolzaei (UvA)，面向服务的协作网络的赋能框架 12 Xixi Lu (TU/e)，在流程挖掘中运用行为语境 13 Seyed Amin Tabatabaei (VUA)，计算可持续的未来 14 Bart Joosten (TiU)，利用时空Gabor滤波器检测社会信号 15 Naser Davarzani (UM)，心力衰竭中的生物标志物发现 16 Jaebok Kim (UT)，自动识别一群儿童的参与度与情绪 17 Jianpeng Zhang (TU/e)，图样本聚类 18 Henriette Nakad (UL)，公证人与私人司法 19 Minh Duc Pham (VUA)，RDF的涌现关系模式 20 Manxia Liu (RUN)，时间与贝叶斯网络 21 Aad Slootmaker (OU)，EMERGO：用于创作和体验基于场景的严肃游戏的通用平台 22 Eric Fernandes de Mello Araújo (VUA)，Contagious：社交网络中行为、感知与情绪传播的建模 23 Kim Schouten (EUR)，语义驱动的基于方面的情感分析 24 Jered Vroon (UT)，用于半自主远程呈现机器人的响应式社会定位行为 25 Riste Gligorov (VUA)，视听资料集中的严肃游戏 26 Roelof Anne Jelle de Vries (UT)，基于理论的量身定制：用于行为改变技术的激励信息 27 Maikel Leemans (TU/e)，用于可扩展软件分析的分层流程挖掘 28 Christian Willemse (UT)，社交触觉技术：它们带来的触感以及它们让你产生的感受 29 Yu Gu (TiU)，普通话语音情绪识别 30 Wouter Beek (VUA)，“语义网”中的“K”代表“知识”：将语义扩展至网络 2019 01 Rob van Eijk (UL)，实时竞价系统中的网络隐私测量：一种基于图的方法，用于RTB系统分类 02 Emmanuelle Beauxis Aussalet (CWI, UU)，用于评估班级规模不确定性的统计与可视化 03 Eduardo Gonzalez Lopez de Murillas (TU/e)，数据库上的过程挖掘：从现实数据源提取事件数据 04 Ridho Rahmadi (RUN)，从临床数据中寻找稳定因果结构 05 Sebastiaan van Zelst (TU/e)，利用流数据进行流程挖掘 06 Chris Dijkshoorn(VUA)，通过众包细分改善关联文化遗产数据集的访问 07 Soude Fazeli (TUD)，社会学习平台中的推荐系统 08 Frits de Nijs (TUD)，~~资源受限的多智能体马尔可夫决策进程~~ ~~09 Fahimeh Alizadeh Moghaddam~~ (UvA)，软件系统的能源效率自适应

10 Qing Chuan Ye (EUR)，用于分配和预测的多目标优化方法 11 Yue Z hao (TUD)，用于理解大规模开放在线课程中学习者行为参与度的学习分析技术 12 Jacqueline Heinerman (VUA)，携手共进 13 Guanliang Chen (TUD)，大规模开放在线课程分析：学习者建模与内容生成 14 Daniel Davis (TUD)，大规模学习分析：对学习者行为建模并改进大规模开放在线课程中的学习成果 15 Erwin Walraven (TUD)，受约束和部分可观测环境下的不确定性规划 16 Guangming Li (TU/e)，基于对象中心行为约束（OCBC）模型的流程挖掘 17 Ali Hurriyetoglu (RUN)，从微文本中提取可操作信息 18 Gerard Wagenaar (UU)，敏捷团队沟通中的人工制品 19 Vincent Koeman (TUD)，用于开发认知智能体的工具 20 Chide Groenouwe (UU)，促进技术增强型人类集体智能 21 Cong Liu (TU/e)，软件数据分析：架构模型发现与设计模式检测 22 Martin van den Berg (VUA)，利用企业架构改进 IT 决策 23 Qin Liu (TUD)，智能控制系统：学习、解释与验证 24 Anca Dumitrache (VUA)，分歧中的真相——面向自然语言处理的众包标注数据 25 Emiel van Miltenburg (VUA)，（自动）图像描述中的语用因素 26 Prince Singh (UT)，同步多式联运集成平台 27 Alessandra Antonaci (OU)，应用于（大规模）开放在线课程的游戏化设计流程 28 Esther Kuindersma (UL)，获准起飞：利用基于游戏的学习为航空公司飞行员应对关键情境做好准备 29 Daniel Formolo (VUA)，利用虚拟智能体在安全关键情境中开展社交技能模拟与训练 30 Vahid Yazdanpanah (UT)，多智能体工业共生系统 31 Milan Jelisavcic (VUA)，生机勃勃：机器人学的点滴探索 32 Chiara Sironi (UM)，面向游戏中通用人工智能的蒙特卡洛树搜索 33 Anil Yaman (TU/e)，人工神经网络中生物启发式学习的演化 34 Negar Ahmadi (TU/e)，用于癫痫和 PNES 分类的脑电微状态与功能性脑网络特征 35 Lisa Facey-Shaw (OU)，利用数字徽章对编程学习进行游戏化 36 Kevin Ackermans (OU)，设计视频增强型评分量规以掌握复杂技能 37 Jian Fang (TUD)，基于现场可编程门阵列的数据库加速 38 Akos Kadar (OU)，学习视觉基础的多语言表征

2020 01 Armon Toubman (UL)，计算出的动作：生成空战行为 02 Marcos de Paula Bueno (UL)，利用概率图模型揭示时间过程 03 Mostafa Deghani (UvA)，利用不完美监督进行语言理解学习 04 Maarten van Gompel (RUN)，语境作为语言桥梁 05 Yulong Pei (TU/e)，局部与全局结构挖掘 06 Preethu Rose Anish (UT)，在需求引出过程中激发架构思维——一种方法与工具支持 07 Wim van der Vegt (OU)，面向可复用游戏组件的软件架构 08 Ali Mirsoleimani (UL)，面向蒙特卡洛树搜索的结构化并行编程 09 Myriam Traub (UU)，衡量工具偏差并改进数字人文研究的数据质量 10 Alifah Syamsiyah (TU/e)，用于流程挖掘的数据库内预处理 11 Sepideh Mesbah (TUD)，面向长尾实体识别模型的语义增强训练数据扩充方法 12 Ward van Breda (VUA)，电子心理健康中的预测建模：探索其在个性化抑郁症治疗中的适用性 13 Marco Virgolin (CWI)，用于遗传编程的基因池最优混合进化算法的设计与应用 14 Mark Raasveldt (CWI/UL)，将分析与关系数据库集成 15 Konstantinos Georgiadis (OU)，Smart CAT：用于严肃游戏中可配置评估的机器学习 16 Ilona Wilmont (RUN)，概念建模的认知方面 17 Daniele Di Mitri (OU)，多模态导师：来自多模态体验的自适应反馈 18 Georgios Methenitis (TUD)，不确定性市场中的智能体交互与机制：可再生能源系统中的电力市场 19 Guido van Capelleveen (UT)，工业共生推荐系统 20 Albert Hankel (VUA)，将绿色信息通信技术成熟度融入组织 21 Karine da Silva Miras de Araujo (VUA)，机器人在哪里？：可能的生活形态 22 Maryam Masoud Khamis (RUN)，通过建模方法理解复杂系统的实施：以桑给巴尔的电子政务为例 23 Rianne Conijn (UT)，写作的关键：利用击键记录，通过写作分析方法研究写作过程 24 Lenin da Nóbrega Medeiros (VUA/RUN)，人类，你感觉如何？：迈向情感支持型聊天机器人 25 Xin Du (TU/e)，异常模型挖掘中的不确定性 26 Krzysztof Leszek Sadowski (UU)，GAMBIT：基于模型的混合整数优化遗传算法 27 Ekaterina Muravyeva (TUD)，教育语境中的个人数据与知情同意

28 Bibeg Limbu (TUD)，用于刻意练习的多模态交互：借助增强现实训练复杂技能 29 Ioan Gabriel Bucur (RUN)，因果推断中的贝叶斯思维 30 Bob Zadok Blok (UL)，创造、更加创造、最具创造力 31 Gongjin Lan (VUA)，更好地学习——从婴儿迈向更好 32 Jason Rhuggenaath (TU/e)，在线市场中的收益管理：定价与在线广告 33 Rick Gilsing (TU/e)，在商业模式创新的背景下支持服务主导型商业模式评估 34 Anna Bon (UM)，干预还是协作？重新设计用于发展的信息与通信技术 35 Siamak Farshidi (UU)，软件生产中的多准则决策 2021 01 Francisco Xavier Dos Santos Fonseca (TUD)，公共空间中用于社会互动的基于位置的游戏 02 Rijk Mercuur (TUD)，模拟人类日常活动：将社会实践理论融入基于智能体的模型 03 Seyyed Hadi Hashemi (UvA)，对用户与智能设备交互进行建模 04 Ioana Jivet (OU)，爱上我的仪表板：为自我调节学习设计自适应学习分析 05 Davide Dell'Anna (UU)，自主系统的数据驱动监督 06 Daniel Davison (UT)，“嘿，机器人，你怎么看？”儿童如何与社交机器人一起学习 07 Armel Lefebvre (UU)，面向开放科学的科研数据管理 08 Nardie Fanchamps (OU)，感知-推理-行动编程对计算思维的影响 09 Cristina Zaga (UT)，机器人事物的设计：通过游戏促进儿童协作的非拟人化、非语言机器人 10 Quinten Meertens (UvA)，统计学习中的错误分类偏差 11 Anne van Rossum (UL)，机器人视觉中的非参数贝叶斯方法 12 Lei Pi (UL)，中国中小企业对外部知识的吸收 13 Bob R. Schadenberg (UT)，面向自闭症儿童的机器人：理解并促进可预测性，以提升学习参与度 14 Negin Samaeemofrad (UL)，企业孵化器：其支持的影响 15 Onat Ege Adali (TU/e)，通过商业服务范式将价值主张转化为资源重新配置 16 Esam A. H. Ghaleb (UM)，基于视听线索的双模态情绪识别 17 Dario Dotti (UM)，利用深度神经网络从运动和身体线索中理解人类行为 18 Remi Wieten (UU)，连接非正式意义建构工具与形式系统：促进贝叶斯网络和论证框架的构建 19 Roberto Verdecchia (VUA)，架构技术债务：识别与管理 20 Masoud Mansoury (TU/e)，理解并缓解推荐系统中的多侧曝光偏差

21 Pedro Thiago Timbó Holanda (CWI)，渐进式索引 22 Sihang Qiu (TUD)，对话式众包 23 Hugo Manuel Proença (UL)，用于预测与描述的稳健规则 24 Kaijie Zhu (TU/e)，高效的时序子图查询处理 25 Eoin Martino Grua (VUA)，电子健康的未来在于移动端：结合人工智能与自适应，打造自适应电子健康移动应用 26 Benno Kruit (CWI/VUA)，解读网格：从人类可读表格扩展知识库 27 Jelte van Waterschoot (UT)，个性化与私人对话：设计希望与您建立联系的智能体 28 Christoph Selig (UL)，理解企业创业项目的异质性 2022 01 Judith van Stegeren (UT)，角色扮演视频游戏的风味文本生成 02 Paulo da Costa (TU/e)，数据驱动的预测与物流优化：深度学习之旅 03 Ali el Hassouni (VUA)，一天一个模型，医生远离我：用于个性化医疗的强化学习 04 Ünal Aksu (UU)，跨组织流程挖掘框架 05 Shiwei Liu (TU/e)，采用即时过参数化的稀疏神经网络训练 06 Reza Refaei Afshar (TU/e)，实时竞价中的广告发布商机器学习 07 Sambit Praharaj (OU)，衡量不可衡量之物？迈向自动化协同空间协作分析 08 Maikel L. van Eck (TU/e)，用于智能产品设计的流程挖掘 09 Oana Andreea Inel (VUA)，理解事件：以多样性为驱动的人机协作方法 10 Felipe Moraes Gomes (TUD)，考察协同搜索引擎的有效性 11 Mirjam de Haas (UT)，在儿童与机器人交互中保持参与：定量研究学龄前儿童在第二语言辅导期间与机器人及任务互动时的参与度 12 Guanyi Chen (UU)，中文名词短语的计算生成 13 Xander Wilcke (VUA)，多模态知识图谱上的机器学习：在真实世界异构空间导向知识上进行学习的机遇、挑战与方法 14 Michiel Overeem (UU)，低代码平台的演变 15 Jelmer Jan Koorn (UU)，流程中的工作：利用流程挖掘发掘意义 16 Pieter Gijsbers (TU/e)，自动化机器学习研究系统 17 Laura van der Lubbe (VUA)，通过严肃游戏和游戏化赋能弱势群体 18 Paris Mavromoustakos Blom (TiU)，玩家情感建模与电子游戏个性化 19 Bilge Yigit Ozkan (UU)，网络安全成熟度评估与标准化

20 Fakhra Jabeen（VUA），数字媒体的黑暗面——社交媒体上负面人类行为的计算分析 21 Seethu Mariyam Christopher（马斯特里赫特大学），用于身体与认知评估的智能玩具 22 Alexandra Sierra Rativa（TiU），虚拟角色设计及其在电子游戏和虚拟现实模拟中促进共情、沉浸感与协作技能的潜力 23 Ilir Kola（TUD），助力支持代理实现社会情境感知 24 Samaneh Heidari（乌得勒支大学），具有社会规范和价值观的智能体——基于社会规范和个人价值观的智能体社会模拟框架 25 Anna L.D. Latour（莱顿大学），约束与不确定性下的最优决策 26 Anne Dirkson（莱顿大学），从患者论坛中发现知识：从患者体验中获取新颖的医学见解 27 Christos Athanasiadis（马斯特里赫特大学），视频序列中的情感感知跨模态领域自适应 28 Onuralp Ulusoy（乌得勒支大学），协作系统中的隐私 29 Jan Kolkmeier（特温特大学），从头部变换到思维移植：混合现实中的社会交互 30 Dean De Leo（CWI），稀疏数组上的动态图分析 31 Konstantinos Traganos（TU/e），借助先进的制造过程管理应对智能制造中的复杂性 32 Cezara Pastrav（乌得勒支大学），社会生态系统的社会模拟 33 Brinn Hekkelman（CWI/TUD），智能电网拥塞管理的公平机制 34 Nimat Ullah（VUA），审视你的行为：用于行为改变的情绪与欲望调节计算建模 35 Mike E.U. Ligthart（VUA），塑造儿童—机器人关系：实现可持续交互的交互设计模式

2023年 01 Bojan Simoski（VUA），解开数字健康干预之谜 02 Mariana Rachel Dias da Silva（TiU），扎根还是在飞翔？我们的身体能告诉我们思想身在何处 03 Shabnam Najafian（TUD），用于群体推荐中隐私保护解释的用户建模 04 Gineke Wiggers（莱顿大学），影响的相关性：结合文献计量学的法律信息检索 05 Anton Bouter（CWI），面向大规模实值优化的最优混合进化算法，包括现实世界中的医学应用 06 AntÃ³nio Pereira Barata（莱顿大学），用于风险评估的~~可靠且公平的机器学习~~ ~~07 Tianjin Huang（TU/e），对抗样本对深度学习可信性~~的作用 08 Lu Yin（TU/e），运用心理测量学习引出知识 09 Xu Wang（VUA），运用语义技术进行科学数据集推荐

10 Dennis J.N.J. Soemers（马斯特里赫特大学），为通用博弈学习状态-动作特征 11 Fawad Taj（VUA），迈向激励机器：为有效的数字健康行为改变应用计算建模行为机制 12 Tessel Bogaard（VUA），利用元数据理解数字图书馆中的搜索行为 13 Injy Sarhan（乌得勒支大学），用于知识表示的开放信息抽取 14 Selma Čaušević（TUD），通过自组织增强能源韧性 15 Alvaro Henrique Chaim Correia（TU/e），关于学习易处理概率图模型的见解 16 Peter Blomsma（TiU），构建具身对话代理：将人类非言语行为观察作为开发人工角色的资源 17 Meike Nauta（特温特大学），可解释人工智能与可解释计算机视觉——从监督到洞见 18 Gustavo Penha（TUD），对话式搜索与推荐模型的设计和诊断 19 George Aalbers （ TiU ），心灵的数字痕迹：利用智能手机捕捉个人幸福感信号 20 Arkadiy Dushatskiy（TUD），采用基于模型的进化算法进行昂贵优化，并将其应用于基于深度学习的医学图像分割 21 Gerrit Jan de Bruin（莱顿大学），交通领域智能检查的网络分析方法 22 Alireza Shojaifar（乌得勒支大学），意志性网络安全 23 Theo Theunissen（乌得勒支大学），持续软件开发中的文档编制 24 Agathe Balayn （ TUD ），机器学习危险故障诊断的实践探索 25 Jurian Baas（乌得勒支大学），历史知识图谱上的实体解析 26 Loek Tonnaer（TU/e），基于线性对称性的解耦表示及其分布外行为 27 Ghada Sokar（TU/e），在变化的数据分布下持续学习 28 Floris den Hengst（VUA），学习如何行动：人类环境中的强化学习 29 Tim Draws（TUD），理解网络搜索结果中的观点偏差 024 01 Daphne Miedema（TU/e），学习 SQL：拆解数据系统教育中的概念 02 E mile van Krieken（VUA），神经符号学习系统中的优化 03 Feri Wijayanto （RUN），用于 Rasch 分析和中介分析的自动模型选择 04 Mike Huisman（莱顿大学），理解深度元学习 05 Yiyong Gou（马斯特里赫特大学），空中机器人作业：多环境协同检查与建筑裂缝自主修复 06 Azqa Nadeem（TUD），通过可解释人工智能理解对手行为：利用序列聚类提取威胁情报

07 Parisa Shayan（TiU），学习管理系统中的用户行为建模 08 Xin Zhou（UvA），从赋能到激励：通过流程设计和激励实施增强策略执行 09 Giso Dal（UT），使用分区贝叶斯网络进行概率推断 10 Cristina-Iulia Bucur（VUA），Linkflows：迈向科学领域真正的语义出版 11 已撤回 12 Peide Zhu（TUD），迈向面向学习的稳健自动问题生成 13 Enrico Liscio（TUD），通过混合智能进行上下文特定价值推断 14 Larissa Capobianco Shimomura（TU/e），图生成依赖关系及其在数据剖析中的应用 15 Ting Liu（VUA），直觉洞见：用于关联肠道微生物组与心理健康的生物医学知识图谱 16 Arthur Barbosa CÃ¢mara（TUD），搜索即学习系统的设计 17 Razieh Alidoosti（VUA），伦理感知的软件架构设计 18 Laurens Stoop（UU），以数据为基础理解能源气象变异及其对能源系统运营的影响 19 Azadeh Mozafari Mehr（TU/e），多视角一致性检查：识别并理解异常行为模式 20 Ritsart Anne Plantenga（UL），规则的运用 21 Federica Vinella（UU），众包以用户为中心的团队 22 Zeynep Ozturk Yurt（TU/e），超越常规：通过可控动态上下文扩展业务流程管理，以适应知识密集型流程 23 Jie Luo（VUA），拉马克的复仇：后天获得性状的遗传改善机器人进化 24 Nirmal Roy（TUD），探索交互式界面对用户网络搜索行为的影响 25 Alisa Rieger（TUD），致力于在网络搜索中对争议话题形成负责任的观点 26 Tim Gubner（CWI），使用 VOILA 框架自适应生成异构执行策略 27 Lincen Yang（UL），用于可解释机器学习的信息论分区模型 28 Leon Helwerda（UL），掌控软件：了解 Scrum 冲刺和待办事项列表的开发进度 29 David Wilson Romero Guzman（VUA），优良、高效与归纳偏置：通过使用归纳偏置探索深度学习中的效率 30 Vijanti Ramautar（UU），模型驱动的可持续性会计 31 Ziyu Li（TUD），探讨元数据在优化机器学习工作流中的效用 32 Vinicius Stein Dani（UU），流程挖掘的阿尔法与欧米伽 33 Siddharth Mehrotra（TUD），为人机交互中的适当信任进行设计

34 Robert Deckers（VUA），从最小软件粒子到系统规范——MuDForM：多领域形式化方法 35 Sicui Zhang（TU/e），使用流程挖掘检测临床偏差的方法：模糊集方法 36 Thomas Mulder（TU/e），图上递归查询的优化 37 James Graham Nevin（UvA），数据处理对计算模型的影响 38 Christos Koutras（TUD），面向现代场景的表格模式匹配 39 Paola Lara Machado（TU/e），商业模式与运营模式之间的联系：从概念理解到可操作的指导 40 Montijn van de Ven（TU/e），指导商业模式关键绩效指标的定义 41 Georgios Siachamis（TUD），流式数据流引擎的适应性 42 Emmeke Veltmeijer（VUA），小群体，大洞见：通过表达性子群分析理解群体 43 Cedric Waterschoot（荷兰皇家艺术与科学院梅尔滕斯研究所），建设性难题：促进在线新闻平台建设性评论的计算方法 44 Marcel Schmitz（荷兰开放大学），迈向学习分析支持的学习设计 45 Sara Salimzadeh（TUD），生活在人工智能时代：了解塑造人机决策的情境因素 46 Georgios Stathis（莱顿大学），预防争议：预防性逻辑、法律与技术 47 Daniel Daza（VUA），利用知识图谱中的子图和属性进行表征学习 48 Ioannis Petros Samiotis（TUD），古典音乐作品的群体辅助标注 2025年 01 Max van Haastrecht（UL），关于有效性的跨学科视角：弥合技术增强型学习系统设计与实施之间的鸿沟 02 Jurgen van den Hoogen（JADS），使用卷积神经网络进行时间序列分析 03 Andra-Denis Ionescu（TUD），以数据为中心的人工智能特征发现 04 Rianne Schouten（TU/e），层次数据的异常模型挖掘 05 Nele Albers（TUD），用于戒烟情境化虚拟辅导的心理学指导强化学习 06 Daniël Vos（TUD），决策树学习：用于稳健预测和策略优化的算法 07 Ricky Maulana Fajri（TU/e），迈向更安全的主动学习：应对不受欢迎的偏差、图结构数据、对抗者和数据不平衡 08 Stefan Bloemheuvel（TiU），通过图进行时空分析：预测建模与图构建 09 Fadime Kaya（VUA），去中心化治理设计——一种基于模型的方法

10 Zhao Yang（莱顿大学），提升目标条件强化学习的自主性与效率 11 Shahin Sharifi Noorian（TUD），从识别到理解：通过多模态语义集成丰富视觉模型 12 Lijun Lyu（TUD），神经信息检索中的可解释性 13 Fuda van Diggelen（VUA），机器人需要一些教育：进化机器人学中学习的复杂性 14 Gennaro Gala（TU/e），基于潜变量层级结构的概率生成建模 15 Michiel van der Meer（莱顿大学），通过混合智能实现观点多样性 16 Monika Grewal（TUD），用于医学影像关键点检测、分割和多目标可变形配准的深度学习 17 Matteo De Carlo（VUA），真实机器人复现：迈向不断进化的机器人生态系统 18 Anouk Neerincx（乌得勒支大学），关怀型机器人：社交机器人如何提升儿童的心理健康 19 Fang Hou（乌得勒支大学），软件生态系统中的信任 20 Alexander Melchior（乌得勒支大学），政策建模不止于政策建模（基于智能体的建模在复杂政策过程中的实用应用） 21 Mandani Ntekouli（马斯特里赫特大学），弥合精神病理学中的个体与群体视角：使用生态瞬时评估数据的计算建模方法 22 Hilde Weerts（TU/e），解读算法公平性：迈向对算法决策中的公平性与歧视的跨学科理解 23 Roderick van der Weerdt（VUA），物联网测量知识图谱：将物联网测量数据构建、处理并学习为知识图谱 24 Zhong Li（莱顿大学），面向智能制造的可信异常检测 25 Kyana van Eijndhoven（TiU），故障中的故障：压力情境下的多层级团队协调动态 26 Tom Pepels（马斯特里赫特大学），蒙特卡洛树搜索仍在持续发展 27 Danil Provodin（JADS, TU/e），复杂反馈下的序贯决策 28 Jinke He（TUD），探索学习得到的抽象模型，以实现高效规划与学习 29 Erik van Haeringen（VUA），百感交集：模拟群体中的情绪传染 30 Myrthe Reuver（VUA），视角之谜：用于负责任新闻推荐的跨学科语言技术 31 Gebrekirstos Gebreselassie Gebremeskel（RUN），聚焦推荐系统：对推荐流程中若干选定组件的贡献 32 Ryan Brate（乌得勒支大学），措辞至关重要：带有情感色彩的词语计算工具包 33 Merle Reimann（VUA），说同一种语言：人机代理与人机交互中的口语能力通信 34 Eduard C. Groen（乌得勒支大学），基于众包的需求工程

35 Urja Khurana（VUA），从概念到影响：推动更稳健的语言模型部署 36 Anna Maria Wegmann（乌得勒支大学），表达相同，方式不同：文体变异与释义改写的计算方法 37 Chris Kamphuis（RUN），探索用于信息检索的关系与图 38 Valentina Maccatrozzo（VUA），打破信息茧房：偶遇发现的语义模式 39 Dimitrios Alivanistos（VUA），用于假设生成的知识图谱与Transformer模型：人工智能时代的科学发现加速 40 Stefan Grafberger（UvA），通过逻辑查询计划实现声明式机器学习流程管理 41 Mozhgan Vazifehdoostirani（TU/e），利用流程灵活性改善流程结果——从描述性分析到可行动洞察 42 Margherita Martorana（VUA），无数据表格的语义解释：面向可发现、可访问、可互操作和可重用的受限访问数据的元数据驱动方法 43 Krist Shingjergji（荷兰开放大学），感知课堂——利用人工智能检测并响应在线教育中以学习为中心的情感状态 44 Robbert Reijnen（TU/e），利用深度强化学习进行机器调度的动态算法配置 45 Anjana Mohandas Sheeladevi（VUA），以居住者为中心的能源管理：平衡智能建筑中的隐私、幸福感与可持续性 46 Ya Song（TU/e），用于工业决策中时间和空间维度建模的图神经网络 47 Tom Kouwenhoven（莱顿大学），协作式意义建构：人类、机器及人机交互中新型语言的涌现 48 Evy van Weelden（TiU），在飞行训练中整合虚拟现实与神经生理学 49 Selene BÃ¡ez SantamarÃa（VUA），具有学习驱动力、以知识为中心的对话代理 50 Lea Krause（VUA），语境化对话式人工智能 51 Jiaxu Zhao（TU/e），理解并缓解生成式语言模型中的不良偏见 52 Qiao Xiao（TU/e），用于神经网络高效训练的模型、数据与通信稀疏性 53 Gaole He（TUD），迈向有效的人机协作：促进对人工智能系统的适当依赖 54 Go Sugimoto（VUA），缺失的关联：考察文化遗产与数字人文领域关联数据及其工具的质量 55 Sietze Kai Kuilman（TUD），闪闪发光的人工智能并非真金：人工智能系统有意义控制的要求 56 Wijnand van Woerkom（乌得勒支大学），举重明轻的基于案例推理：人工智能与法律应用中的形式化研究

57 Syeda Amna Sohail（UT），医疗保健元数据共享中的隐私-效用权衡及其他：组织间与组织内层面的规范性与实证评估 58 Junhan Wen（TUD），《从图像到市场》：机器学习赋能的水果供应 59 Mohsen Abbaspour Onari（TU/e），从解释到信任：可解释决策支持中的信任建模与测量 60 Marcel Jurriaan Robeer（UU），超越信任：执法领域可解释人工智能的因果方法 61 Shuai Wang（VUA），大型集成知识图谱中的关联：分析、完善与领域应用 62 Khaleel Asyraaf Mat Sanusi（OU），在沉浸式学习环境中增强心理运动技能学习模型 63 Rashid Zaman（TU/e），退化数据上的在线一致性检查 64 Jens d’Hondt（TU/e），高效且有效的多变量相似性搜索 65 Aswin Balasubramaniam（UT），揭示跑步者与无人机交互的潜能 2026 01 Pei-Yu Chen（TUD），人类-智能体对齐对话：在运行时获取用户信息，以支持个性化行为 02 Hezha Hassan Mohammedkhan（TiU），基于二维图像估算儿童身体测量值：迈向营养不良自动检测 03 Kyriakos Psarakis（TUD），云应用规模化的普及：流式数据流上的事务性有状态函数