面向短时轻量级流的无服务器抽象

Natalie Carl
柏林工业大学 柏林，德国
nc@3s.tu-berlin.de

Niklas Kowallik
柏林工业大学 柏林，德国
nk@3s.tu-berlin.de

Constantin Stahl
柏林工业大学 柏林，德国
cs@3s.tu-berlin.de

Trever Schirmer
柏林工业大学 柏林，德国
ts@3s.tu-berlin.de

Tobias Pfandzelter
柏林工业大学 柏林，德国
tp@3s.tu-berlin.de

David Bermbach
柏林工业大学，德国柏林
db@3s.tu-berlin.de

arXiv:2603.03089v1 [cs.DC] 3 Mar 2026

摘要

无服务器计算和流处理是事件驱动数据处理的两种主流范式，但二者都基于一些假设，导致它们在处理需要有状态处理的短时、轻量且不可预测的数据流时效率低下。我们提出流函数，将其作为函数即服务模型的一种新扩展，把短数据流作为执行、状态和扩缩容的单元。流函数通过基于迭代器的接口处理数据流，从而实现无缝的事件间逻辑，同时保留无服务器平台提供的弹性和缩至零能力。我们的评估表明，在一个视频处理用例中，与成熟的流处理引擎相比，流函数可将处理开销降低 ~99%。流函数在提供与无服务器函数相当的性能和流语义的同时，为现有模型难以满足的一类工作负载提供了有效且高效的抽象。

关键词

函数即服务、无服务器计算、流处理

1 引言

函数即服务 (FaaS) 是一种基于事件的云计算模型，开发者在其中编写小型无状态函数，以处理单个离散事件。无服务器计算平台负责函数生命周期的各个方面：从底层资源管理，到函数执行环境、函数调用和函数实例调度 [14]。因此，开发者可以主要专注于应用逻辑，而按使用量计费则省去了大量固定的基础设施开销。这种范式已应用于构建 REST API、处理来自物联网设备的数据、响应数据库更新，以及运行定期安排的任务和 cron 作业等场景 [8]。

可以说，轻松扩缩容的能力是函数即服务的主要卖点。一方面，无状态性带来了内在弹性：无服务器平台维护着动态的函数实例池，这些实例可互换地用于处理传入请求。函数实例会根据负载和配置被创建或移除——进而通过缩容至零来减少空闲资源使用和成本。由此产生的弹性扩缩容能力，使函数即服务应用程序能够近乎实时地处理高度突发性工作负载 [13]。

然而，这并非没有代价。由于无服务器平台牢牢掌控着调度，因此无法保证两个请求由同一实例处理。因此，保持状态需要外部存储，或函数实例之间的直接通信。前者缓慢且成本高昂，而公共无服务器平台缺乏针对后者的一等机制 [6]。这将函数即服务应用程序限制在跨事件计算不频繁或完全无关的用例中。更具体地说，无服务器函数不适合处理带有事件间处理逻辑的连续数据流。

尽管人们已提出多种有状态无服务器计算方案，但这些方案大多无法或无意与专为此目的设计的成熟流处理引擎竞争 [19]。有状态无服务器计算平台为函数配备键值存储，以便在多次函数调用之间保存状态 [24]。然而，这仍要求无服务器平台针对数据流中的小片段调度、调用并扩缩函数实例，而不是在网络连接打开期间让函数保持运行，通过该连接在整个数据流期间持续发送数据。相比之下，流处理引擎专为处理长时间运行的连续数据流而设计，具有较高但相对稳定的吞吐量和较低的延迟。开发者将计算表示为由数据流连接的算子图，这些算子预先部署，并且无论数据何时到达都会保持活跃 [2]。通过将初始化和资源分配成本分摊到持续运行的流水线生命周期中，这种模型能够实现高效的事件间处理、顺序保证和有状态计算。然而，正如我们将要展示的，这些假设限制了无服务器应用所具备的灵活性和弹性。具体而言，对于短暂或零星出现的数据流，流处理引擎效率低下：当数据流到达时间不可预测或频率较低时，启动开销和始终开启的资源分配会占据执行时间的主要部分。

本文认为，存在一类流处理应用，当前的函数即服务和流处理方法都不适用于它们。具体而言，这类应用有四个特征。(1) 数据流**运行时间短**，通常只有几秒或几分钟。(2) 它们是**轻量级的**，吞吐量为每秒数兆字节。(3) 它们的到达时间**不可预测**。(4) 应用程序必须能够以**有状态的**方式处理数据流。综合来看，这些特征揭示了当前函数即服务产品和流处理引擎的不足。虽然无服务器函数能够处理运行时间短、

轻松应对轻量级且不可预测的请求，但在输入持续不断且需要有状态处理时却显得力不从心 [19]。相比之下，流处理引擎擅长处理持续运行的任务，却难以快速缩容至零，并且只能在需要时使用资源 [23]。因此，函数即服务与流处理之间存在一个缺口，目前还没有合适的编程抽象来填补。

为此，我们概述了此类应用的不同示例（§2），比较无服务器计算与流处理抽象（§3），通过扩展函数即服务模型，提出将流函数作为一种替代方案（§4），并将我们的概念验证实现与不同的流处理和无服务器计算替代方案进行比较（§5）。

2 动机示例

以下示例旨在说明，为了高效处理运行时间短暂的短暂流，需要一种合适的编程抽象。虽然每个示例都 可以借助现有技术以不同方式解决，但我们指出了其中的不足，以说明为何需要新的抽象。

示例 *1*：按需视频处理。 设想由社交媒体平台上的用户交互触发的按需视频预处理流水线。与旨在处理大量连续到达数据的传统媒体流水线不同，这些工作负载处理的是时长仅数秒、吞吐量适中的短视频流。此类数据流的到达具有内在的不确定性，取决于用户行为或外部触发因素。考虑到此类工作负载可能每天或每周才出现，持续运行视频处理流水线来应对这种零星负载，会导致利用率低下和不必要的成本。同时，使用无状态无服务器函数处理单独的帧，会使帧间操作变得复杂，因为这些操作需要在不同帧之间保留状态。延迟往往至关重要：预处理通常处于内容分发、分析或后续处理阶段等下游任务的关键路径上。因此，系统必须快速实例化处理逻辑，在视频片段的整个处理期间维护每个流的状态，并在处理完成后立即释放资源。现有流处理引擎假设流水线长期运行，并通过长时间运行来摊销启动成本；而无服务器平台缺乏将执行和状态绑定到短视频流生命周期的抽象。因此，按需视频预处理流水线是传统流处理与无服务器计算之间工作负载的典型例子。在 §5中，我们将使用此用例的简化版本进行评估。

示例 *2*：自然灾害监测。科学监测系统利用分布式传感器网络（包括地震仪、河流水位计和气象站）监测地震、洪水或火山活动等自然灾害。虽然数据会持续采集，但通常只有在满足特定条件时（例如阈值超限或检测到异常）才会触发详细分析。触发后，系统会流式传输一小段持续数秒至数分钟的传感器数据，以便立即处理。处理过程中需要在数据流持续期间维护状态，例如计算滚动统计数据、跟踪近期基线，或关联不同传感器的测量值。鉴于此类物理事件十分罕见，这些数据流的到达时间本质上无法预测。为所有传感器持续运行分析流水线效率低下，而使用无状态无服务器函数处理单个测量值则不可行。因此，合适的系统必须能够快速扩展，以处理有界数据流及每个流的状态，并在处理完成后立即缩减规模。现有的流处理引擎和无服务器计算平台都无法满足这些要求，因此需要探索其他编程抽象。

示例 *3*：交互式数据分析。在交互式分析系统中，用户经常发起查询，触发短暂的数据突发流，以便立即进行处理，例如探索性聚合或转换。这些数据流生命周期很短，到达时间不可预测，通常需要在多条记录之间维护中间状态，并且需要尽可能快速地处理。为这类零星交互配置长期运行的分析流水线效率低下，而逐事件无服务器执行则会使有状态处理变得复杂。因此，支持交互式分析需要处理启动迅速且能立即缩容的短时有状态数据流。

3 无服务器计算与流处理编程模型

现代云平台将函数即服务和流处理作为处理事件驱动数据的两种主流编程模型。尽管二者在实践中都得到广泛应用，但它们对执行、状态和扩缩容的基本假设截然不同。

无服务器计算。函数即服务平台采用事件驱动编程模型，即针对每个传入事件独立调用函数。平台负责调度、扩缩容和隔离，使应用程序能够根据需求弹性扩展。函数实例默认无状态，也不保证相关事件由同一个实例处理。因此，无服务器平台扩展的是事件，而非数据流。事件之间的顺序不会自动保留，任何事件间逻辑都需要借助外部状态，或通过函数实例之间的显式通信来实现。尽管这种设计能够针对独立请求实现细粒度弹性和高效的资源使用，但对于计算过程自然跨越多个相关事件的应用程序（例如连续数据流或基于会话的数据流），它会增加处理难度。

流处理。 流处理引擎采用面向流水线的编程模型，开发者在其中定义由数据流连接的长时间运行的算子图。事件间逻辑通过算子和窗口构造来表达，这些构造根据时间或其他标准对事件进行分组。这些系统通过并行化算子来扩展规模，并通过检查点和重放提供可靠的处理保证。根据平台的不同，算子可以通过声明式的类 SQL 语言指定，也可以通过高级编程语言实现自定义处理逻辑。该模型假定流水线会预先部署，并且无论数据是否到达都会持续运行。启动成本和资源使用会分摊到长时间运行的工作负载中，因此流处理引擎非常适合处理持续、高吞吐量的数据流

。然而，正如我们将要论证的那样，这些假设使它们在处理短暂或零星出现的数据流时效率低下，因为初始化开销会占据大部分执行时间。

衔接尝试。 有多个系统尝试将无服务器弹性与流语义结合起来，例如，为无服务器函数添加有状态存储，或以弹性、按需的方式部署流处理框架。尽管这些方法降低了一些开销，但通常仍保留事件逐个处理的无服务器执行或基于流水线的流处理这两种核心抽象。因此，现有模型缺乏一流支持，无法将短生命周期流作为执行、扩缩容和状态生命周期的单元。这促使我们需要一种编程抽象，将流语义与无服务器平台的弹性、按使用付费特性结合起来。

4 流函数

我们提出流函数，作为对函数即服务模型的一种全新扩展。我们认为，对于以资源高效的方式处理数量高度波动的短小轻量级流而言，流函数提供了最自然的编程抽象。

4.1 编程模型：迭代器输入，迭代器输出

在无服务器计算中，函数签名充当平台与用户代码之间的接口，并决定了编程抽象。每个函数接收一个独立事件作为输入，例如 HTTP 请求或云事件，并可选择返回响应。相比之下，流函数接收一个迭代器作为输入，并可选择返回一个迭代器。与仅为单个事件指定逻辑相比，函数代码决定了如何处理整个 Stream。清单1 展示了 Go 编程语言中的流函数。开发者提供函数 Fn，并将其部署到无服务器平台。Stream 由一系列事件组成，每个事件包含字节负载和元数据，例如时间戳及其他元数据。通过遍历输入流，可以处理事件，并按需表达事件间逻辑。在此示例中，开发者调用 yield 函数，将新事件输出到其他处理步骤。

与传统的函数即服务函数类似，处理逻辑易于实现，而且函数具有可移植性。在
部署时，用户会为处理后的事件指定输出。将多个流函数串联成一个工作流，即可构建任意复杂的处理流水线。每个流函数都对应工作流中的一个处理步骤。因此，应用程序图不会进行优化——这也使得新的流函数实例可以快速启动或销毁，而不受其他处理步骤的影响。对多个数据流进行并行处理，类似于使用多个函数实例处理多个请求。与传统的函数即服务函数类似，实例可以通过进程、容器或微型虚拟机等方式实现隔离 [1, 25]。此外，迭代器是大多数高级编程语言中的一等组件。虽然我们使用 Go 进行说明，但这一概念可以轻松适配其他常用于无服务器计算的语言，从而实现无缝采用。

```
type Event struct { Headers map[string]any
Payload []byte } func Fn(stream iter.Seq[Event])
iter.Seq[Event] { return func(yield func(Event)
bool) { for e :=range stream { // ... } } }
```

列表 1：流函数及类型定义的 Go 语言框架。用户提供的函数 Fn 会遍历数据流中的所有元素，并按需调用 yield 函数返回输出。

4.2 函数实例详解

如图 1所示，流函数实例由三个组件构成：(1) 用户提供的函数，(2) 先进先出消息缓冲区，以及 (3) 函数处理器，它将输入从消息缓冲区传递给用户函数，并处理可选输出。函数处理器和用户提供的函数位于同一进程中，以尽量降低二者之间传递消息的开销。其目的是对数据流本身的底层处理提供抽象，并限制用户对缓冲区的访问。队列充当一个小型缓冲区，仅用于在初始化期间提供额外存储空间，以及在处理事件暂时超过平均到达间隔时间时提供余量。

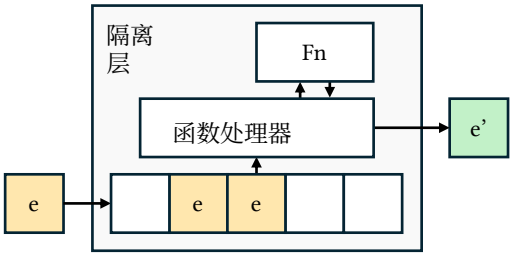

图 1：流函数实例的组件。事件在由函数处理程序传递给函数之前会先进行缓冲。转换后的事件会根据配置发出。

所有组件都封装在容器中，以隔离不同函数实例。请注意，我们有意省略无服务器平台设计的细节，只关注函数实例的编程模型和结构。因此，根据具体实现，可以互换使用不同的消息传递和隔离机制。

4.3 函数生命周期

流函数实例的生命周期包含三个阶段。(1) 流生产者一旦与平台建立连接，控制平面就会创建一个新实例，初始化消息缓冲区，并启动配备了用户函数的函数处理器。(2) 在整个 Stream 持续期间，函数实例会持续处理数据，并（可选地）发出处理后的事件。重要的是，用户函数在初始化期间只会接收一次函数调用。之后，函数处理器会通过调用用户函数时传入的迭代器，向其提供新事件。(3) 流生产者与平台之间的网络连接一旦关闭，实例就会关闭，并释放为缓冲区分配的内存。请注意，为了降低冷启动延迟，流函数的平台实现可以让实例保持预热状态，以便日后复用。默认情况下，Stream 结束后，我们会立即缩容至零。

5 评估

流函数旨在大幅降低处理短小、轻量级数据流时系统初始化的开销。为此，我们将概念验证实现与三种替代方案进行比较，以此作为基准。

5.1 视频处理用例

我们实现了一个负载生成器，以每秒 10 帧的速率生成图像数据流，分辨率为 160×120。该数据流由 20 张随机生成且彼此不同的图像循环组成，这些图像不会被处理步骤缓存。对于每个系统，我们都提供一个适配器，使用首选协议传输数据流。单个处理步骤会将每帧转换为灰度图，方法是将每个像素的 RGB 值设为其平均值。

原型与基线。 我们实现了概念验证原型 Beaver，并将其以开源形式发布。它包含一个控制平面，用于接受传入的网络连接并处理函数生命周期（§4.3），以及一个如上所述用于视频处理的函数实例。该函数实例与消息缓冲区（我们使用 Nats）一同被封装在 Docker 容器中以实现隔离。作为基线一（以下简称“Stream”），我们使用 Apache Beam [10] 与 Google Cloud Dataflow [2] 运行器实现相同的处理步骤。Beam 管道包含一个 PTransform，用于从 Google Cloud Pub/Sub 读取帧。初步实验表明作业提交速度较慢。为此，我们大幅延长 Pub/Sub 确认期限，使其长于实验持续时间，以免遗漏任何帧。接下来，基线二（“函数即服务”）是一个 Google Cloud Run Function (v2)，由负载生成器针对流的每一帧进行调用。我们将最大实例数限制为 1。最后，基线三（“Batch”）同样实现为 Cloud Run Function (v2)；不过，负载生成器会先创建完整的流，并只调用一次该函数，将所有帧作为输入。由于所有帧都必须放入函数内存，我们有意将负载生成器配置为使用低分辨率。所有系统均使用 Go 编程语言实现。

实验设置。 所有系统均运行在 Google Cloud 平台的 europe-west10 区域。负载生成器与 Beaver 分别运行在独立的 e2-standard-2 虚拟机上，每台配备 2 个虚拟 CPU 和 8 GB 内存。除非另有说明，我们使用默认设置以确保可复现性。对于两个 Cloud Run Functions，我们在各次运行之间留出足够时间，以便先前的所有实例都被销毁。

附注。 请注意，此用例违反了 §1 中定义的第四项特征，因为将图像流转换为灰度图并不涉及状态。这是有意为之的简化，目的是在不受特定状态存储解决方案影响的情况下测量函数即服务基线。此外，我们在所有测量值中都计入冷启动时长。我们承认，流处理引擎并未针对快速冷启动进行优化，而这正是我们要强调的：当到达时间不可预测时，持续运行流处理引擎的成本会高得令人难以承受。因此，能够应对 §1 中所定义特征的系统必须考虑这一点。

5.2 性能指标

我们定义了两项性能指标，用于将 Beaver 与基线一至三进行比较。首先，*冷启动开销* 用于衡量实验时长中用于冷启动的比例，如公式（1）所示。我们关注以下三个时间戳。t_0 表示创建第一帧的时间点，t_1 表示处理系统接收第一帧的时间点，而 t_2 表示处理完最后一帧的时间点。在极端情况下，θ 值为 1 意味着整个处理时长都耗在冷启动上。反之，随着θ趋近于零，用于冷启动的时间占比也随之趋近于零。总体而言，冷启动开销衡量的是冷启动使流处理延迟了多少。

$$\theta = \frac{t_1 - t_0}{t_2 - t_0} \tag{1}$$

接下来，*处理开销* 衡量的是总耗时（$t_2 - t_0$）与理论上最快可能的处理耗时之间的差值。这个下限由基准测试客户端生成 Stream 所需的时间决定，因为在事先不了解数据的情况下，不可能比生成 Stream 更快地处理它。在我们的实验中，这分别对应 10 s、30s 和 60 s。

5.3 结果

图 2 展示了不同系统和流持续时间下冷启动开销的测量值。由此可见，系统分为两组：Beaver 和函数即服务的 θ 值接近于零（持续时间为10 秒 时分别为 0.0363 和 0.0391，持续时间为60 秒时分别为 0.0061 和 0.0062）；而 Stream 和 Batch 的值接近于 1（持续时间为10 秒 时分别为 0.9981 和 0.9759，持续时间为60 秒时分别为 0.9575 和 0.9764）。随着流持续时间增加，Stream 基线的改善幅度最大。Stream 和 Batch 基线的相似性表明，尽管它们处理数据的意图和方法不同，但其行为出人意料地相似。在 Batch 的情况下，处理步骤会有意等待，直到流完全生成后才开始处理数据。虽然 Stream 基线的抽象假定事件会持续到达，但缓慢的冷启动

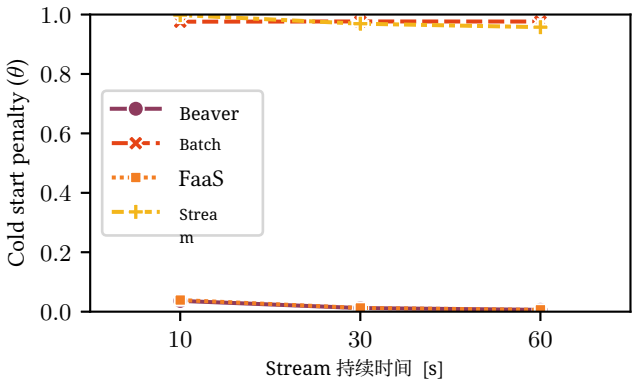

图 2：不同配置与数据流持续时间下的冷启动开销。由于冷启动很快，Beaver 与函数即服务方法的 θ 值接近于零。随着数据流变长，Batch 方法迅速改善，但代价是首次处理前的等待时间增加。由于冷启动较慢，流处理方法不适合短数据流。

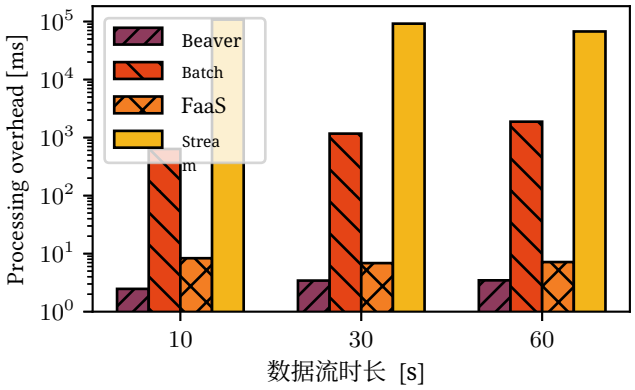

图 3：相对于理论最小值的处理开销。随着数据流变长，批处理开销会增加，而 Beaver、函数即服务和 Stream 的开销则略有下降。平均而言，与 Stream 方法相比，Beaver 可将处理开销降低约 99%，其表现与函数即服务方法相当。

会导致所有帧都积聚在消息系统中。冷启动完成时，所有数据都已完全生成，Stream 实际上就成了一个批处理系统。

图 3 展示了相对于理论下限的处理开销。由此可以清楚地看出各实现的排序：Stream 实现最慢，耗时约 107.84s；其次是 Batch，耗时 636.25 ms。函数即服务和 Beaver 实现的性能相近，耗时均不到 10 ms。随着数据流持续时间增加，函数即服务和Beaver 实现的处理开销始终保持稳定。虽然两者的冷启动耗时都在一百多毫秒，但当帧间平均时间超过平均处理时间后，它们都能追上输入流。在输入流结束时，开销变得至关重要，两者都已准备好预热实例，以极低的开销处理最后一帧。由于负载生成器和受测系统都位于同一数据中心，网络延迟很低。尽管Beaver 的表现略胜于函数即服务，但这种差异很可能源于生产系统与研究原型之间的区别：每一帧都需要经过多层负载均衡器，才能到达函数实例。正如预期，Batch 的开销会随着数据流持续时间的增加而增加。由于只有在数据流完成后才开始处理，输入数据量增加也会延长处理所需的时间。对于较长的输入，Stream 的表现有明显改善。虽然冷启动耗时相近（117.584s、118.088s和 121.896s），但我们会减去数据流的长度，以得到处理开销。随着输入流变长，快速处理在一定程度上抵消了冷启动的固定成本。对于短数据流，这清楚地表明，大型流处理引擎并不适用于我们的使用场景，因此需要采用其他方法。不过，随着数据流持续时间增加，开销会有所改善，这表明不同系统之间可能存在一个转折点。

6 讨论与展望

我们的结果表明，无服务器抽象非常适合处理轻量级流。无服务器平台和流处理引擎都针对各自的工作负载进行了充分优化，但都无法很好地适应短小、轻量的数据流。因此，我们在评估中观察到的行为，直接源于这些编程模型中内含的假设。

6.1 恰到好处?

我们认为，流函数解决了设计空间中一个鲜少探索的领域。当数据流运行时间短、轻量、不可预测且需要有状态处理时，流函数能够在函数即服务力有不逮、流处理又未针对其优化的领域中，提供具备无服务器特性的流语义。不过，流函数并非适用于所有场景的解决方案。相反，它们代表了一种互补的方法，适用于当前的函数即服务和流处理系统都不太适合的设计空间区域。我们的评估表明，对于具备上述特征的工作负载，流函数成功地提供了流处理抽象，同时具备无服务器计算的优势。

我们有意将采用 Google Cloud Dataflow 运行器的 Apache Beam 作为基线，因为它代表了生产级流处理系统的当前最高水平。我们的目标并非展示配置错误或效率低下，而是评估成熟的流处理引擎在面对短生命周期、不可预测的数据流时的表现。所观察到的冷启动开销，是优先考虑吞吐量、容错和长期运行流水线的设计选择所带来的直接结果。因此，这些结果凸显了现有流处理抽象与§1中所定义的工作负载特征之间的根本不匹配，而非 Dataflow 本身的局限性。

6.2 顺序、状态与故障语义

流函数提供的执行语义有意与短生命周期流的生命周期保持一致。每个传入的数据流由单次函数调用处理，从而自然地保留事件的到达顺序，并按流隔离状态。与传统函数即服务平台不同——在后者中，事件被独立调度，并可能由不同实例乱序处理——流函数将顺序和状态绑定到一条流的持续期间，而非单个事件。

流函数中的状态仅在数据流的生命周期内存在。如果处理过程中发生故障，所有内存状态都会丢失，除非应用程序明确将其外部化。虽然这意味着其容错保证弱于流处理引擎所提供的保证，但我们认为，这种设计选择符合目标工作负载的特征。对于短时运行的轻量级流，检查点和重放的成本很容易超过其收益；不过，未来的无服务器平台若有需要，可以轻松添加此功能。这与流处理引擎形成了鲜明对比：流处理引擎针对长期运行的流水线进行了优化，并通过检查点、重放和精确一次处理等机制提供强有力的保证。流函数则以这些保证换取快速启动、逐流隔离和弹性资源使用。

6.3 垂直扩展

当单个数据流的吞吐量超出单个流函数实例的处理能力时，我们的设计便会受到限制。流函数可以随着并发流生产者数量的增加自然扩展，但它们本身并不支持对单个数据流的处理进行并行化。

传统无服务器函数通过复制无状态函数实例并进行负载均衡来实现可扩展性。当实例之间无需协调时，事件可以自由分发，无需考虑顺序或共享状态。流函数在处理多个数据流时保留了这一优势，但对单个数据流进行有状态处理，会带来更严格的扩缩容限制。在这种情况下，要提高吞吐量，就需要将数据流分片到多个函数实例上，这会立即引发与顺序保证和状态一致性相关的挑战。流函数纵向扩缩容的难度，还因有状态流处理任务的多样性而进一步加剧。例如，计算移动平均值时，可以先对事件进行轮询分区，再执行聚合；而视频处理工作负载则可能需要对单独的帧进行空间分区。这些策略有着本质区别，无法仅凭流抽象自动推导出来。因此，流函数不存在普遍适用的分片策略。

未来的无服务器平台可以通过多种方式解决这一局限。一种方案是将垂直扩缩容限制在定义明确的狭窄场景中，这类似于某些现有流处理引擎如何通过按显式键分区来并行化算子。或者，平台可以向开发者开放分片和合并逻辑，让他们进行应用专用分区，但代价是增加复杂性。这带来了一个有趣的挑战，我们计划在未来的工作中系统地解决这一问题。

7 相关工作

关于事件驱动数据处理的先前研究大致分为两类：流处理系统和无服务器计算平台。二者都关注可扩展数据处理的重要方面，但其设计基于对执行生命周期、状态管理和弹性的不同假设。尽管有各种研究尝试将无服务器计算模型拓展至流处理领域，反之亦然，但我们认为，流函数涉及这两个领域之间一个独特的方向：处理天然面向流、但运行时间短、轻量且不可预测的工作负载，正如§1中所述。

流处理系统专为长期运行的连续流水线而设计，通过随时间分摊启动成本和资源分配，在持续的高吞吐量工作负载上表现出色。然而，这些系统假定数据流持续处于活跃状态，因此难以灵活应对快速波动或不可预测的工作负载。已有若干工作将无服务器原则融入流处理以提升弹性，例如将无服务器函数与流处理引擎结合使用，以避免过度配置 [23]。另有工作将算子与其流处理引擎解耦，使算子能够动态修改，而无需重启整个流水线 [17]。尽管这些方法缓解了特定局限，但它们大体上仍将基于流水线的执行作为主要抽象。由于其关注点不同，短生命周期流并未被当作具有明确定义的生命周期、状态范围和扩展行为的一等实体来对待。

相比之下，无服务器平台采用逐事件执行模型，从而实现细粒度弹性和快速缩容至零的行为。这种灵活性的代价是失去顺序保证和无缝的事件间逻辑，因为平台既无法控制由哪个实例处理给定事件，也无法直接寻址函数实例 [11]。大量工作通过为函数配备键值存储或共享日志，以在多次调用之间保留状态 [4,12, 18, 22, 24] ，或通过调度、函数组合或显式消息传递来增强实例之间的协调与通信 [3, 6, 7, 9, 15, 20, 21]，从而应对这些局限。此外，更丰富的触发机制允许函数一次性响应多个相关事件，从而减少对显式状态管理的需求，为有状态执行提供了一种互补方法 [5, 16]。尽管取得了这些进展，现有系统仍将独立调度的函数调用作为其核心抽象。因此，它们扩缩容的是单个事件而非数据流，并且对在持续时间未知的连续数据流上表达事件间逻辑仅提供有限支持。

8 结论

本文提出了流函数这一编程抽象，面向介于传统流处理与无服务器计算之间的一类工作负载。我们的目标是流导向、但运行时间短、轻量且不可预测的应用——这些特征以不同方式违背了这两种模型的假设。流函数将流语义与无服务器平台的弹性生命周期管理相结合，把短生命周期流本身视为执行单元、

状态和扩缩容的单元。我们认为，通过使执行和状态生命周期与数据流的持续时间保持一致，流函数为处理短暂流提供了自然的基础。

我们使用 Beaver（概念验证实现）进行的评估表明，这种抽象可以显著降低短时运行流的开销。与成熟的流处理系统相比，Beaver 在保持与逐事件无服务器执行相当性能的同时，将处理开销降低了 ~99%。这些结果表明，现有系统中观察到的低效现象并非主要源于实现细节，而是源自工作负载特征与底层编程模型之间的根本性不匹配。我们认为，流函数凸显了重新思考无服务器系统执行粒度的重要性。随着云应用日益结合事件驱动行为与流处理行为，将执行和状态绑定到有意义的工作单元上的抽象变得至关重要。流函数是迈向此类抽象的第一步，并为构建高效、弹性且富有表现力的数据处理系统开辟了新的机会。

参考文献

[1] Alexandru Agache、Marc Brooker、Alexandra Iordache、Anthony Liguori、Rolf Neugebauer、Phil Piwonka 和 Diana-Maria Popa。2020。Firecracker：面向无服务器应用的轻量级虚拟化。载于第 17 届 USENIX 网络系统设计与实现研讨会（美国加利福尼亚州圣克拉拉）（NSDI '20）。USENIX 协会，美国加利福尼亚州伯克利，419–434。https://www.usenix.org/ conference/nsdi20/presentation/agache [2] Tyler Akidau、Robert Bradshaw、Craig Chambers、Slava Chernyak、Rafael J. Fernández-Moctezuma、Reuven Lax、Sam McVeety、Daniel Mills、Frances Perry、Eric Schmidt 和 Sam Whittle。2015。数据流模型：在大规模、无界、乱序数据处理中平衡正确性、延迟与成本的实用方法。VLDB 会刊 8，12（2015），1792–1803。doi:10.14778/2824032.2824076[3] Istemi Ekin Akkus、Ruichuan Chen、Ivica Rimac、Manuel Stein、Klaus Satzke、Andre Beck、Paarijaat Aditya 和 Volker Hilt。2018。SAND：迈向高性能无服务器计算。载于 2018 年 USENIX 年度技术会议（美国马萨诸塞州波士顿）（ATC '18）。USENIX 协会，美国加利福尼亚州伯克利，923–935。https://www.usenix.org/conference/atc18/presentation/akkus [4] Daniel Barcelona-Pons、Marc Sánchez-Artigas、Gerard París、Pierre Sutra 和 Pedro García-López。2019。踏上函数即服务之路：利用无服务器架构构建有状态分布式应用。载于第 20 届 Middleware 国际会议论文集（美国加利福尼亚州戴维斯）（Middleware '19）。美国计算机协会，美国纽约州纽约，

41 54 10 1145 3361525 3361535 – 。doi: 。/ 。[5] Natalie Carl、Trever Schirmer、Niklas Kowallik、Joshua Adamek、Tobias Pfandzel- ter、Sergio Lucia 和 David Bermbach。2025。面向无服务器计算的多事件触发器。载于 2025 年 IEEE 国际云工程会议（法国雷恩）（IC2E '25）。IEEE，皮斯卡塔韦，美国，146–154。doi:10.1109/IC2E65552. 2025.00030[6] Marcin Copik、Roman Böhringer、Alexandru Calotoiu 和 Torsten Hoefler。2023。FMI：面向无服务器函数的快速、低成本消息传递。载于第 37 届 ACM 国际超级计算会议论文集（美国佛罗里达州奥兰多）（ICS '23）。美国计算机协会，美国纽约州纽约，373–385。doi:10.1145/3577193.3593718[7] Vojislav Dukic、Rodrigo Bruno、Ankit Singla 和 Gustavo Alonso。2020。Photons：精简版 Lambda 函数。载于第 11 届 ACM 云计算研讨会论文集（美国虚拟会议）（SoCC '20）。美国计算机协会，美国纽约州纽约，45–59。doi:10.1145/3419111.3421297[8] Simon Eismann、Joel Scheuner、Erwin van Eyk、Maximilian Schwinger、Johannes Grohmann、Nikolas Herbst、Cristina L. Abad 和 Alexandru Iosup。2022。无服务器应用现状：收集、特征分析与社区共识。IEEE 软件工程汇刊 48，10（2022），4152–4166。doi:10.1109/TSE.2021.3113940[9] Sadjad Fouladi、Riad S. Wahby、Brennan Shacklett、Karthikeyan Vasuki Balasubra- maniam、William Zeng、Rahul Bhalerao、Anirudh Sivaraman、George Porter 和 Keith Winstein。2017。编码，快与慢：使用数千个微型线程进行低延迟视频处理。载于第 14 届 USENIX 网络系统设计与实现研讨会（美国马萨诸塞州波士顿）（NSDI '17）。USENIX 协会，美国加利福尼亚州伯克利，363–376。https://www.usenix.org/conference/ nsdi17/technical-sessions/presentation/fouladi [10] Apache软件基金会。2016。统一的Apache Beam模型。https: //beam.apache.org/。（访问日期：2026年1月26日）。[11] Joseph M. Hellerstein、Jose Faleiro、Joseph E. Gonzalez、Johann Schleier-Smith、Vikram Sreekanti、Alexey Tumanov 和 Chenggang Wu。2019。无服务器计算：前进一步，后退两步。载于第 9 届创新数据系统研究会议论文集（美国阿西洛马）（CIDR '19）。美国计算机协会，美国纽约州纽约，9 页。https://www.cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf[12] Zhipeng Jia 和 Emmett Witchel。2021。Boki：基于共享日志的有状态无服务器计算。载于 ACM SIGOPS 第 28 届操作系统原理研讨会论文集（德国虚拟会议）（SOSP '21）。美国计算机协会，美国纽约州纽约，

691 707 10 1145 3477132 3483541 – 。doi: 。/ 。[13] Eric Jonas、Qifan Pu、Shivaram Venkataraman、Ion Stoica 和 Benjamin Recht。2017。占领云端：为 99% 人群提供的分布式计算。载于 2017 年云计算研讨会论文集（美国加利福尼亚州圣克拉拉）（SoCC '17）。美国计算机协会，美国纽约州纽约，445–451。doi:10.1145/3127479.3128601[14] Eric Jonas、Johann Schleier-Smith、Vikram Sreekanti、Chia-Che Tsai、Anurag Khandelwal、Qifan Pu、Vaishaal Shankar、Joao Carreira、Karl Krauth、Neeraja Yadwadkar、Joseph E. Gonzalez、Raluca Ada Popa、Ion Stoica 和 David A. Patterson。2019。简化云编程：伯克利对无服务器计算的看法。arXiv: 1902 03383 [ ] 1902 03383 。cs.OS

[15] 2026 https://arxiv.org/abs/ 。Niklas Kowallik、Trever Schirmer 和 David Bermbach。 。Konflux：面向无服务器应用的优化函数融合。arXiv:2601.11156 [cs.DC] https: // arxiv.org/abs/2601.11156[16] Pedro García López、Aitor Arjona、Josep Sampé、Aleksander Slominski 和 Lionel Villard。2020。Triggerflow：无服务器工作流的基于触发器的编排。载于第 14 届 ACM 分布式与事件驱动系统国际会议论文集（加拿大魁北克省蒙特利尔）（DEBS '20）。美国计算机协会，美国纽约州纽约，

3 14 10 1145 3401025 3401731 – 。doi: 。/ 。[17] Manisha Luthra、Sebastian Hennig、Kamran Razavi、Lin Wang 和 Boris Kolde- hofe。2020。算子即服务：有状态无服务器复杂事件处理。载于 2020 年 IEEE 国际大数据会议（美国虚拟会议）（Big-Data '20）。IEEE，皮斯卡塔韦，美国，1964–1973。doi:10.1109/BigData50022.2020. 9378142[18] Tobias Pfandzelter 和 David Bermbach。2023。Enoki：从边缘到云的有状态分布式函数即服务。载于第 2 届边缘中间件国际研讨会论文集（意大利博洛尼亚）（MiddleWEdge '23）。美国计算机协会，美国纽约州纽约，19–24。doi:10.1145/3630180.3631203[19] Tobias Pfandzelter、Sören Henning、Trever Schirmer、Wilhelm Hasselbring 和 David Bermbach。2022。流处理与函数：云端事件处理的成本视角。载于 2022 年 IEEE 国际云工程会议（美国加利福尼亚州太平洋丛林市）（IC2E '22）。IEEE，皮斯卡塔韦，美国，67–78。doi:10. 1109/IC2E55432.2022.00015[20] Qifan Pu、Shivaram Venkataraman 和 Ion Stoica。2019。洗牌，快与慢：无服务器基础设施上的可扩展分析。载于第 16 届 USENIX 网络系统设计与实现研讨会（美国马萨诸塞州波士顿）（NSDI '19）。USENIX 协会，美国加利福尼亚州伯克利，193–206。https://www.usenix.org/ conference/nsdi19/presentation/pu [21] Trever Schirmer、Joel Scheuner、Tobias Pfandzelter 和 David Bermbach。2024。FUSIONIZE++：通过动态任务内联和基础设施优化提升无服务器应用性能。IEEE 云计算汇刊

12, 4 2024 1172 1185 10 1109 2024 3451108 （ ），– 。doi: 。/TCC. 。[22] 2020 Simon Shillaker 和 Peter Pietzuch。 。FAASM：用于高效有状态无服务器计算的轻量级隔离。载于 2020 年 USENIX 年度技术会议（ATC '20）。USENIX 协会，美国加利福尼亚州伯克利，419–433。https: //www.usenix.org/conference/atc20/presentation/shillaker [23] Won Wook Song、Taegeon Um、Sameh Elnikety、Myeongjae Jeon 和 Byung-Gon Chun。2023。Sponge：借助无服务器框架实现流处理的快速响应式扩缩。载于 2023 年 USENIX 年度技术会议（美国马萨诸塞州波士顿）（ATC '23）。USENIX 协会，美国加利福尼亚州伯克利，301–314。https: // www.usenix.org/conference/atc23/presentation/song [24] Vikram Sreekanti、Chenggang Wu、Xiayue Charles Lin、Johann Schleier-Smith、Joseph E. Gonzalez、Joseph M. Hellerstein 和 Alexey Tumanov。2020。Cloudburst：有状态函数即服务。VLDB 会刊 13，12（ 2020 2438 2452 10 14778 3407790 3407836 ），– 。doi: 。/ 。[25] Ethan G. Young、Pengfei Zhu、Tyler Caraza-Harter、Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。2019。隔离的真实成本：gVisor 案例研究。载于第 11 届 USENIX 云计算热点研讨会（美国华盛顿州伦顿）（HotCloud '19）。USENIX 协会，美国加利福尼亚州伯克利，6 页。https://www.usenix.org/conference/hotcloud19/presentation/young