Cuttlefish: A Fair, Predictable Execution Environment for Cloud-hosted Financial Exchanges

Liangcheng Yu
Microsoft Reserach and University of Pennsylvania, USA

Prateesh Goyal
Microsoft Reserach, USA

Ilias Marinos
Nvidia, USA

Vincent Liu
University of Pennsylvania, USA

Abstract

Recent years have seen a rising interest in cloud-hosted financial exchanges. While the public cloud platforms promise a cost-effective and more accessible option to traders, unfortunately, achieving fairness in cloud environments is challenging due to non-deterministic network latencies and execution times.

This work presents Cuttlefish, a fair-by-design cloud execution environment for algorithmic trading. The idea behind Cuttlefish is the efficient and robust mapping of real operations to a novel formulation of 'virtual time.' With it, Cuttlefish abstracts out the variances of the underlying network communication and computation hardware. Our implementation and evaluation not only validate the practicality of Cuttlefish, but also show its operational efficiency on public cloud platforms.

2012 ACM Subject Classification Networks → Cloud computing; Networks → Network dynamics; Applied computing → Online auctions

Keywords and phrases Cloud-hosted exchanges, Financial exchanges, Computation and communication variances, Virtual time overlay

Digital Object Identifier 10.4230/LIPIcs.AFT.2025.3

Related Version *Full Version*: `https://www.microsoft.com/en-us/research/publication/cuttlefish` [58]

1 Introduction

Low-latency algorithmic trading—a subset of the broader securities trading taxonomy—is pivotal to the efficiency of modern financial markets, promoting accurate and timely pricing of securities, higher liquidity, and lower trade costs for all investors [32]. The goal of low-latency algorithmic trading is to create an ecosystem within each exchange where, based on incoming market data, traders can issue buy and sell orders as quickly as possible to take advantage of ephemeral market-making and arbitrage opportunities [57]. A sizable fraction of activity in today's exchanges is the result of this class of trades [2, 24].

In recent years, exchanges like NASDAQ, CME, LSE, and B3, alongside cloud providers like Microsoft, Amazon, and Google, have expressed growing interest in exploring the viability of hosting this type of trading in the cloud [22, 49, 42, 41]. Their interest is rooted in a variety of factors including better scaling, fewer outages, improved cost savings, and a potentially broader customer base as, in this model, any Market Participant (MP)[1] can rent a machine

[1] In low-latency algorithmic trading, MP also refers to the computer program executing the trading algorithm, terms we use interchangeably.

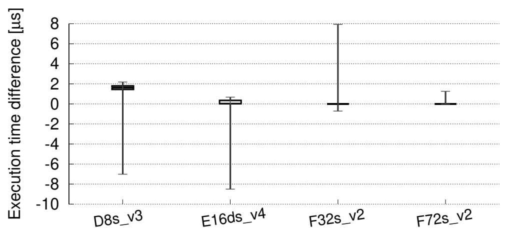


Figure 1 Execution time differences (candlesticks showing $p0.1$, $Q1$, $Q2$, $Q3$, and $p99.9$, with some quantiles visually converging to a single bar) measured with `rdtsc` for 1M parallel invocations of two MPs running on separate VM instances within the same region, despite an identical program (performing a moving average crossover strategy, with each invocation taking ≈ 14µs on D8s_v3), input market data stream, performance optimizations, and VM type.

in the same region as the Central Exchange Server (CES) and participate, bypassing the logistical hurdles involved in installing and maintaining on-premise hardware [24, 17, 19, 18]. Smaller and newer exchanges (e.g., for cryptocurrency) are particularly interested, as the cloud can also lower their barriers to entry by eliminating the need for custom infrastructure.

Unfortunately, in addition to the above advantages, cloud-hosted exchanges also face significant challenges in ensuring fairness for MPs, a primary requirement of exchange services [48, 24, 17, 18]. Recent work has noted these exchanges' issues with unpredictable network latencies [24, 38, 19], but the sources of unfairness extend beyond the network and include practicalities like noisy neighbors, thermal fluctuations, or (un)scheduled maintenance [39, 53, 55]. For example, Figure 1 shows the differences between local execution time of two identical trading programs running on parallel instances in the same Azure region. The experiment is for a simple program, single-tenancy (in the case of the `F72_v2` instance type), and strict scheduling policies (core pinning, highest kernel scheduling priority, etc.). It further omits network variability and clock drift effects—two important and difficult-to-control sources of unfairness. Even so, we can observe significant variability and bias between the two executions.

Crucially, the magnitude of the variability is immaterial—*any* difference, no matter how small, may alter the order in which trades are processed by the CES. We note that some of these effects also exist in traditional on-premise exchanges; however, cloud-hosting (besides increasing the sources and magnitude of variability/bias) presents a qualitative change in how these effects must be handled. In an on-premise deployment, customers are in full control of their infrastructure—when a machine is slow, it is the customer's fault. In the cloud, infrastructure can be slow through no fault of the customer, and the responsibility falls upon the cloud provider.

In this paper, we tackle an ambitious goal: a cloud execution environment where the outcome of races between different MPs' buys/sells is based *only* on the design of the MPs rather than (un)lucky performance fluctuations of their underlying cloud infrastructure. Our approach is similarly ambitious: to change the execution model from one where users have unfettered access to (virtualized) hardware to one where users provide bytecode-level programs (closer to a Functions-as-a-Service interface) and cloud providers control their rate of execution to ensure fairness.

The resulting system, Cuttlefish[2], is an execution environment for a cloud-hosted exchange

[2] The animal renowned for its ability to see invisible polarized light to discern subtle changes in murky waters for navigation and communication.

that ensures fair, predictable end-to-end execution. Cuttlefish is the first to address execution variation/bias in cloud-hosted exchanges, but its focus on end-to-end guarantees means that it also handles deficiencies in the communication guarantees of prior work that specialize in communication [17, 24, 19]. Further, not only is Cuttlefish able to guarantee these strong properties, but it is able to do so while offering low latency and high throughput.

Guaranteeing this level of fairness is fundamentally challenging as, in the end, simultaneous data delivery and synchronous execution is a classic (and under some assumptions impossible) challenge in distributed systems [20, 13, 56, 24]. Moreover, modern hardware performance is increasingly unpredictable, complex to reason about, and difficult to verify [39, 53, 55, 61, 12, 4, 33, 3, 9].

Cuttlefish achieves the above using an efficient mapping of real operations to 'virtual time' from a platform-agnostic intermediate representation (IR). This *indirection* through virtual time allows us to quantify computation and, critically, to control its advancement deterministically through the rate-limiting of MP operations—a level of control that is not possible with real time. This approach enables deterministic and fair operations in both simultaneous market data release and MP exeuction processes and guarantees fairness, regardless of the varying latencies in the communication of market data and trade responses or variations in execution platforms.

This concept of virtual time mirrors that of other applications such as co-simulation [4, 33, 3, 9], which coordinates virtual time for concurrent emulation and simulation processes. Cuttlefish takes a step further by extending the concept to real-time cloud systems and developing an end-to-end trading platform. Cuttlefish's architecture is, thus, a combination of (1) a platform-agnostic Intermediate Representation (IR) instantiating virtual time per virtual machine instruction cycle count, along with its expressive programming interface and lightweight instrumentation for virtual cycle tracking, (2) a runtime execution environment optimized for co-located MPs, and (3) a protocol to control inflight virtual cycles and handle variations in the underlying network or compute.

While Cuttlefish represents an extreme point in the design space, our prototype demonstrates the feasibility and efficiency of its design, deployable to commercial cloud virtual machines. This paper makes the following contributions:

- We propose Cuttlefish, the first fair execution environment for cloud-hosted exchange that abstracts out the differences in the underlying cloud primitives, tackling execution fairness and simultaneously addressing persistent gaps in the communication fairness of existing systems.
- We introduce an efficient mapping strategy of the virtual time overlay to real-time operations while maintaining low latency and high execution throughput to MPs.
- We evaluate Cuttlefish using an end-to-end implementation on a real cloud platform. When serving 100 MPs, Cuttlefish guarantees fairness in both communication and execution, while introducing low overhead and achieving latency and execution throughput close to the limits of the underlying cloud hardware.

2 Background

Historically, financial exchanges were bustling places where people would shout orders, negotiate prices, and physically exchange papers representing ownership of stocks or other assets. In today's financial markets, however, the vast majority of trades are executed by computers rather than by humans, opening up the possibility of so-called algorithmic trading techniques, now the cornerstone of modern financial markets.

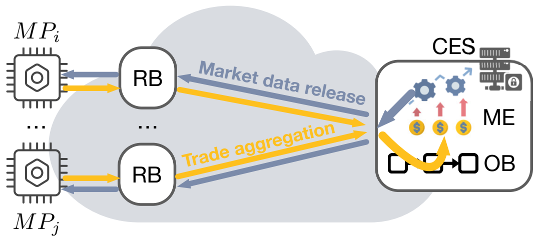


Figure 2 Basic structure of cloud-hosted exchanges. All system components are controlled by the trusted cloud provider.

Algorithmic trading refers to the process of making trade decisions with the help of computer programs. Under this umbrella, low-latency algorithmic trading, which focuses on fast (e.g., <1 ms) reactions to real-time market data with minimal human intervention, has become critical to market efficiency, price discovery, liquidity, and low transaction costs. These algorithms account for a significant portion of the trading volume in financial markets [2, 24, 32, 36, 57]. Compared to the broader set of algorithmic trading strategies, the logic of MPs who participate in low-latency algorithmic trading is relatively simple (designed for quick reactions) and features highly optimized data path logic [37, 34].

Exchanges that support low-latency algorithmic trading expose a simple top-level abstraction: (1) the exchange broadcasts market data to all MPs within the exchange, delivered at time $\{D\}$, (2) the MPs analyze the data, and (3) they return buy/sell orders to the exchange at time $\{S\}$, processed in the order of submission.

In the most cases, the market data feed is the sole input into the system, and orders are handled entirely within the exchange. However, MPs sometimes also integrate outside data into their strategies. Exchanges can also provide alternative trading interfaces [27], although these are typically three to four orders of magnitude higher latency and used by MPs for less latency-critical trading (e.g., those leveraging complex machine learning models). In this work, we assume that all such external interactions are funneled through the CES (in the case of alternative trading interfaces) or a special gateway node (Section 7; in the case of outside data). Clearly, Cuttlefish cannot control the outside world or make it fair, rather, our focus is only on the variability and bias of communication and execution after it enters the system through these nodes. We discuss the interaction of Cuttlefish with the broader ecosystem in Section 11.

Cloud-hosted exchanges. As mentioned, there has been recent interest in cloud-hosted exchanges for reasons including better scaling, fewer outages, improved cost savings, and broader access to the financial markets [24, 19, 17, 36].

Figure 2 depicts the main components of these exchanges. At the core of these systems is the CES, which disseminates market data to all MPs through Release Buffers (RBs) or equivalents. The communication between CES and MPs typically leverages a reliable message transport [24, 19]. Then, the MPs—hosted by proxy cloud instances in the same cloud region as the CES—compute their trading decisions and forward them to the CES.[3] There, trades are first enqueued and sorted at the Ordering Buffer (OB) and then processed by the Matching Engine (ME). The ME finally updates the limit order book and generates a new

[3] Cross-region deployment of VMs is feasible (e.g., through virtual network (VNet) peering [45]), but not suitable for low-latency communication [24, 19].

Notation	Definition
$G(x)$	Wall-clock time when market data x is generated.
$D(i,x)$	Wall-clock time when data x is delivered to MP_i.
$S(i,y)$	Wall-clock time trade y is submitted by MP_i.
$\widetilde{D}(x)$	Virtual time assigned by CES for delivering data x.
$\widetilde{S}(i,y)$	Virtual time when trade y is submitted by MP_i.
$\widetilde{CES}(t)$	Virtual time of CES at a wall clock time t.
$\widetilde{OB}(t)$	Virtual time of OB at a wall clock time t.
$\widetilde{MP_i}(t)$	Virtual time of MP_i at a wall clock time t.
$\widetilde{CES}^{-1}(vt)$	Wall clock time of CES at a virtual time vt.
$\widetilde{OB}^{-1}(vt)$	Wall clock time of OB at a virtual time vt.
$\widetilde{MP_i}^{-1}(vt)$	Wall clock time of MP_i at a virtual time vt.

Table 1 Summary of notations. Each refers to a scalar timestamp in real or virtual time.

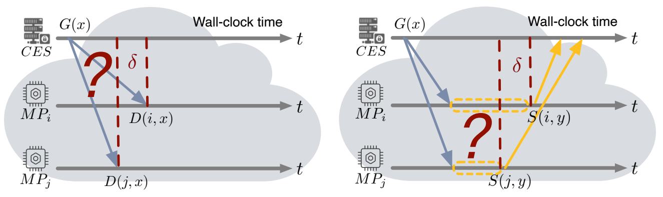
batch of market data.

Performance variations of cloud primitives. Unfortunately, despite their attractive properties, in our conversations with major cloud providers, financial exchanges, and trading firms, there is still a fundamental distrust of the performance properties of the underlying infrastructure. In particular, for both exeuction and communication, performance variations have the potential to invalidate the benefits of careful design, creating a world where MPs win and lose not on the strength of their algorithms but on purely external factors (e.g., temporal variation or provider monitoring/maintenance).

To illustrate, consider the simple scenario of delivering market data x for MP_i and MP_j in Figure 3a. Coordinating delivery to ensure $D(i,x) = D(j,x)$ is difficult due to unpredictable and unbounded path latencies. More importantly, even with simultaneous data delivery, the same hardware substrate (e.g., same server SKUs), the same software stack (OS and the MP's trading algorithm), and the same algorithm, the exeuction time can still vary (e.g., due to different thermal state). This leads to non-deterministic submission times $S(i,y) \neq S(j,y)$, as shown in Figure 3b. In both cases, *any* disparity, even at nanosecond time scales, can advantage/disadvantage an MP.

Recent measurements on cloud-hosted exchanges have quantified the danger of latency spikes in modern clouds [24, 19]. Equally important, however, is the bias and variability in execution performance, which can also occur for any number of reasons, including everything from non-deterministic software operations to machine-specific hardware wear and thermal effects, with prior measurement citing the Coefficient of Variation (CoV) of performance in bare-metal infrastructure of up to 30% [55, 39]. These are on top of noisy neighbor and hypervisor effects introduced in cloud deployments. Our benchmarks on Azure cloud, shown in Figure 1, validate the presence of these biases and variances. Even for single-tenant `F72_v2` instances, with all physical cores of the machine reserved, the interquartile range (IQR) is still 20 ns with disparities exceeding $> 1\,\mu$s at $p99.9$.

While in both cases it may be possible for cloud providers to try to tame this effect, for instance, by removing all management and telemetry infrastructure or carefully controlling temperature and wear, resulting in much tighter SLOs, (1) doing so would substantially cut into the cost and scalability advantages of cloud-hosted exchanges, and (2) this still does not eliminate disparities that can result in unfairness.



(a) Communication unfairness **(b)** Execution unfairness

Figure 3 Simultaneous data delivery and execution fairness are difficult, even under the same market data, MP algorithm, and execution platform.

3 Goals and Related Work

Our goal in this work is to create a fair and predictable execution environment where the outcome of competitions is based on only the properties of MPs' algorithms rather than luck. More specifically, we target:

R1 *Communication fairness:* (*a*) All MPs should get access to the market data points at the same time and (*b*) in the other direction, MP_a's trade should execute before MP_b iff MP_a submits a trade before MP_b.

R2 *Execution fairness:* Given any execution of the algorithm on the platform, the submission time of the generated trades is *completely* defined by the delivery time of the input data to the algorithm. Consequently, given *R1*, for a given trading algorithm, the execution time to generate the trades should not vary across MPs.

To be practical, MPs should be able to sustain the above requirements while maintaining low latency and high execution throughput for their trades and algorithms. As previously noted, we explicitly do not consider the fairness or feasibility of cross-exchange arbitrage, which is likely impossible in a partial deployment scenario.

Prior work on cloud exchange fairness. Existing cloud-hosted exchanges, despite focusing exclusively on communication [24, 17, 19], provide incomplete guarantees, even for *R1*. For example, CloudEx [17] and Octopus [18] enforce high-resolution clock synchronization among all RBs and the CES. The CES, upon generating market data at time t, assigns a future release timestamp $t + \Delta t_r$ with a predefined threshold Δt_r, allowing RBs to forward the data simultaneously. Similarly, when an inbound trade arrives at an RB at time t, the CES enqueues it to the OB until $t + \Delta_d$ where the delay Δ_d allows earlier trades to arrive within this headroom. Unfortunately, even with perfect clock synchronization—a strong assumption in distributed systems [38]—the guarantees break whenever latency spikes exceed the threshold. Such latency spikes can occur unpredictably in cloud environments [24, 19, 35]. Configuring conservative threshold values can help (at the cost of performance) but are not a complete remedy [24, 19, 13].

More recent work, DBO [24, 19], relaxes the requirement of clock synchronization and proposes logical clocks based on MP response time. Briefly, DBO offloads RBs to local SmartNICs that measure and tag each MP's response time, while trades are being ordered accordingly at the OB/CES. This method corrects inaccuracies in simultaneous data delivery post hoc and provably guarantees Limited Horizon Response Time Fairness for of *R1*. However, DBO's guarantees are limited to a specific trading pattern, namely trigger-point-based high-speed trades. Trades that do not fit this model (e.g., trades triggered using two or more data points) are not necessarily fair.

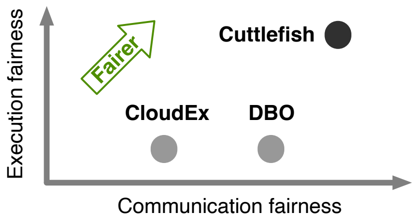


Figure 4 Recent cloud-hosted exchanges [17, 24, 19, 18] target only communication fairness, but still struggle to achieve it. We discuss other related works in Appendix D.

4 Virtual Time in Cuttlefish

Cuttlefish tackles both execution and communication fairness simultaneously. As depicted in Figure 4, we find that including execution fairness not only serves to present a more predictable execution platform, it also naturally addresses the fundamental limitations of existing work on communication.

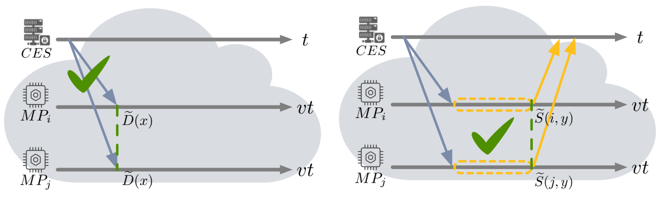
To guarantee both *R1* and *R2*, our system, Cuttlefish, adopts a 'virtual time' abstraction. Virtual time, as a general concept, is not new—there are many instances where it is beneficial to have a global and fine-grained notion of dependencies independent of wall-clock time.

Of particular relevance is the use of virtual time in high-fidelity emulation of processes interacting over a network [3, 33, 9, 4, 23]. In these frameworks, all processes keep a virtual clock for use in coordinating per-process progress and cross-process events, e.g., network communication. Unlike wall-clock time, virtual time is controllable: a process's virtual clock advances only when it is scheduled. Virtual time is, thus, a stand-in for the expected behavior of the emulated network. The framework exploits the ability to pause and resume processes to ensure that all processes are synchronized and events are sequenced correctly according to their virtual time.

Network emulation vs. low-latency algorithmic trading. Cuttlefish takes an analogous approach by assigning virtual time to all communication and execution—down to an instruction level. Like emulation, Cuttlefish benefits from the ability to control the fine-grained progression of virtual time for each MP (pausing and skipping forward as necessary). Unlike emulation, however, low-latency algorithmic trading presents a substantially different set of goals and knobs.

Soft real-time constraints on virtual time progression: Generally, the primary concern of network emulation is fidelity to a target emulated network. The relationship between the emulator's virtual time and wall-clock time is of secondary importance, with the most important impact being its effect on the end-to-end execution time of emulation. In contrast, Cuttlefish is a platform for trading real-world financial instruments, so consistent timeliness is critical, especially in the presence of alternative trading interfaces and external data.

Control over input frequency: Emulation's focus on fidelity also generally assumes a 'correct' emulation target. In contrast, the CES in Cuttlefish has significant control over market data delivery times—what matters is the fairness of the delivery, not fidelity to any particular execution. Cuttlefish uses this control to adjust the market data delivery rate in response to the current load and to allow lagging nodes to catch up.



(a) Communication fairness **(b)** Execution fairness

Figure 5 Communication and execution fairness are achievable in the virtual time domain through the deterministic control of virtual time passage and the quantification of execution.

Cuttlefish virtual time, illustrated. Figure 5 depicts the operation of virtual time in Cuttlefish. The Cuttlefish CES operates unrestricted in *wall-clock time*, while the MPs track and adhere to *virtual time*. For simplicity, we will ignore component failures but discuss how Cuttlefish can be extended to handle them in Section 8. The notation used in the figure and the remainder of the paper are summarized in Table 1.

1. [Figure 5a] For fair market data delivery (*R1a*), the CES picks and tags a virtual time $\widetilde{D}(x)$ for the release of each data point x. Each MP's local execution runtime controller ensures the release accordingly.
2. [Figure 5b] For execution fairness (*R2*), Cuttlefish provides a deterministic accounting of compute (in virtual time). Specifically, it counts the instruction cycles executed by its platform-agnostic virtual machine substrate. This ensures that, for any MP_i consuming an input x at $\widetilde{D}(x)$ and producing a trade y, the resulting $\widetilde{S}(i,y)$ is deterministic, regardless of execution performance variations.
3. [Figure 5b] For fair trade ordering (*R1b*), trades from an MP are marked with the virtual time at which they are generated, $\widetilde{S}(i,y_1)$ and $\widetilde{S}(j,y_2)$. Similar to DBO, Cuttlefish features an ordering buffer that forwards these trades to the CES based on their generation virtual time in monotonically increasing order. Thus, the ME processes y_1 before y_2 if and only if $\widetilde{S}(i,y_1) < \widetilde{S}(j,y_2)$.

Strict adherence to virtual time on all MPs ensures both (R1) and (R2). Further, the CES's ability to bridge wall-clock and virtual time ensures minimal delta between the two (Section 8).

5 Design Overview

Instantiating virtual time abstraction end-to-end necessitates several building blocks: How can we express computation in the virtual time domain (§6.1)? How do we execute the programs efficiently (§7)? How can we track and control the virtual time advancement (§8)?

Architecture. Similar to other cloud-hosted exchanges [24, 19], VMs executing MPs are co-located in the CES region for low-latency services. Unlike existing systems, however, users of Cuttlefish build on a platform-agnostic virtual machine substrate clocked by virtual time cycles, mediating all MPs' computational and I/O operations through this abstraction.

Figure 6 depicts Cuttlefish's high-level architecture. For simplicity, most of our discussion will assume homogeneous, single-threaded market participants and the market data stream as the only input to the ecosystem. We describe the integration of external data in Section 7, extension to heterogeneous compute in Section 11, and multi-threading support in Appendix A.

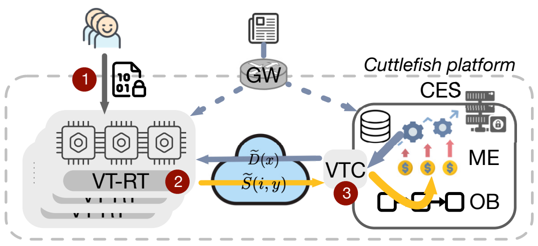


Figure 6 Overview of the Cuttlefish platform.

Workflow. Building on the typical cloud-hosted exchange architecture, Cuttlefish introduces:

❶ *MP algorithm representation via eBPF VM bytecode [Section 6]:* To account for the amount of execution deterministically, Cuttlefish leverages a platform-agnostic IR that is based on the eBPF Virtual Machine (VM)[4] instruction set. Cuttlefish advances virtual time based on the consumed number of VM instruction cycles, allowing it to abstract out potential variances in the underlying infrastructure. Cuttlefish also adapts eBPF user-space libraries to support a simple but expressive programming interface. Cuttlefish verifies, instruments, and translates this code from MPs to native assembly for the underlying computation platform for efficient execution.

❷ *Virtual time execution runtime (VT-RT) [Section 7]:* Cuttlefish develops a runtime execution environment that can efficiently utilize all available cores to execute the binaries for MPs allocated to the same cloud VM. It also manages a range of real-time operations for the responsible MPs, including tracking and advancing the virtual time, data delivery, and local batching of trades and heartbeat to the central CES for ordering based on the submitted virtual time.

❸ *Virtual time control (VTC) [Section 8]:* Cuttlefish integrates a virtual time control algorithm for the CES to assign virtual market data release times. By controlling virtual time assignment, the CES controls how much compute throughput is available to each MP. This is crucial to mitigating underlying network latency spikes or slowdown in execution behaviors.

The cost of fairness. Cuttlefish prioritizes fairness, which is the primary concern for exchanges [48, 24, 19]. This trade-off is intentional, as a slower exchange operating within the higher latency bounds of public clouds can still meet market needs, provided it delivers market data and accepts orders uniformly across all participants [24, 48, 40].

In exchange for fairness, Cuttlefish incurs modest overheads on MP execution. Some of this is due to the extra instrumentation to track and control virtual time (∼2–20%). More fundamentally, guaranteeing that all MPs have equal opportunity means that system-wide progress is gated on the slowest node. This limitation is intrinsic to any fair system. Prior work like DBO and CloudEx noted similar limitations when equalizing network delay; Cuttlefish incurs the same for execution.

Despite these trade-offs, in our evaluation on a public cloud and 100 MPs (approximately the maximum scale of most existing exchanges [24, 27]), Cuttlefish incurs low overhead in both latency and throughput under real-world performance variation, approximating the limits of the underlying cloud hardware (Section 10.3). If higher throughput is needed,

[4] A virtual machinery abstraction, not to be confused with physical cloud VMs.

better hardware or multi-threaded execution (Appendix A) can help. Also, the observed latency is well within the requirements of many major exchanges.[5] Regardless, Cuttlefish still guarantees fairness and predictability in all cases.

Trust assumptions and threat model. Moving to a cloud-hosted solution fundamentally requires MPs to trust the cloud provider and trading platform to not steal or manipulate the MP trading code and execution [17, 19, 24]. Such requirements are typically enforced through contracts, log auditing, privacy laws, and regulatory bodies such as the SEC, and the same prohibitions apply here. Recent advances in hardware (e.g., Intel SGX [10]) as well as efforts in cloud confidential computing [47, 1, 26] could also benefit Cuttlefish by providing cryptographic attestation and secure enclaves during both instrumentation and execution. An exploration of these mechanisms is out of the scope of this work.

For the provider's side of the provider-MP relationship, following prior work [24, 19, 17], we assume that all core system components—including release and ordering buffers—are trusted and protected against tampering. As we discuss in Sections 6.2 and 8, Cuttlefish's abstractions naturally defend against problematic instructions or program structures. Otherwise, MPs can utilize the system in arbitrary ways, including trying to slow down virtual time advancement through interaction with the CES and execution engine.

For all other interactions (e.g., MP-to-MP or to/from malicious third parties), we assume secure and exclusive communication channels between MPs and Cuttlefish components using standard public cloud features, such as security groups [24, 19] to prohibit outside communication with any of these entities. While it is possible that malicious actors may attempt to indirectly influence these connections, e.g., through datacenter-level DDoS attacks, cloud providers themselves are already quite capable of mitigating, defending against, and provisioning for these types of attacks.

6 MP Algorithm Representation

This section elaborates on Cuttlefish's abstractions and platform-agnostic IR, taking a top-down approach.

6.1 Programming Interface

Recall from Section 2, MP algorithms consist of processing CES data feeds to make trading decisions that aim to optimize profit from price disparities, bid-ask spreads, or liquidity subsidies. To allow users to easily program MP algorithms, Cuttlefish utilizes a simple event-driven programming interface.

MP handler abstraction. Figure 7 shows a simplified example of how users may express trading logic with Cuttlefish's *mp_handler* interface.

An MP's handlers are invoked serially on each subscribed market data point. Virtual time advances on every new invocation (in accordance with *R1a*) and on every execution of an IR instruction (with a fixed virtual time cost per instruction).

More specifically, for each market data x, the virtual time of MP_i is updated according to the rule $\widetilde{MP_i} = \max(\widetilde{D}(x), \widetilde{MP_i})$. This involves two scenarios: (1) If the prior invocation finishes before $\widetilde{D}(x)$, Cuttlefish advances $\widetilde{MP_i}$ to the target virtual time and releases the

[5] A major exchange, IEX, prides itself on fairness with a 700μs latency [27, 24].

```
#include <cuttlefish_user.h> /* Single include of whitelist APIs */
int mp_handler (subscribed_context_t* ctx):
    if (ctx->price) > 100 then
        trade_t trade = 1; /* Sell */
        submit_trade(&trade); /* Just-in-time trade submission */
    else if (ctx->price) < 10 then
        trade_t trade = 2; /* Buy */
        submit_trade(&trade);
    update_map(0, &ctx->price); /* Save the history price */
    return 0;
```

Figure 7 An example MP pseudocode in high-level language using **Cuttlefish service APIs**, which includes a narrow interface to a KV store for stateful invocations.

data; and (2) if the MP handler chose to consume more cycles that ends up overshooting $\widetilde{D}(x)$, $\widetilde{MP_i}$ remains unchanged. MPs can submit trades at any point in this process. Each trade's ordering is determined by the exact virtual time of the associated `submit_trade` call.

The cost of each instruction is fixed and public knowledge (for details on the map between individual IR instruction types to its virtual time cost, see Section 9).

eBPF IR. Although any platform-agnostic runtime could serve as a virtual hardware substrate, Cuttlefish chooses an IR based on the eBPF VM instruction set. This IR is compelling for many reasons: it is simple (a 64-bit RISC register machine), it has a mature ecosystem including support for various language frontends, and it is widely supported in multiple target architectures including specialized hardware accelerators (e.g., FPGAs, smartNICs) [5, 31]. More importantly, the simple eBPF Instruction set architecture (ISA) allows us to easily enforce a constrained memory access model and reason about safety by verifying MPs' eBPF bytecode accordingly before execution through static analysis [54, 16, 31].

We note that using the eBPF IR does *not* mean that we are using the kernel-based eBPF VM. While the kernel-based VM [16] imposes restrictions that limit expressiveness, e.g., loop bounds, Cuttlefish does not impose such constraints. Thus, it provides a *Turing-complete* interface [16, 54] for trading strategies, and we show examples of these in Table 4.

Usability: Users can write their trading programs directly in eBPF bytecode (and will likely do so for performance reasons), or they can use more accessible toolchains (such as `llvm`'s eBPF backend) to compile the MP expressed in a high-level language like `C` to the bytecode and then sent (e.g., as an `elf` file) as input to Cuttlefish.

Service APIs: To enable user access to Cuttlefish's trading services, Cuttlefish provides a single header file that contains main data structures and a *whitelist* of shared service APIs. These include: (1) primitive service APIs for trade submission and virtual time facilities, as well as a runtime context object for accessing real-time market data, current virtual time $\widetilde{MP_i}(t)$, and the release virtual time $\widetilde{D}(x)$ for the current invocation, (2) a narrow interface for KV store interactions (e.g., update, lookup) for stateful invocations, and (3) extensible built-in computational helpers like FFT—which users can optionally leverage for convenience—although users can also write their own implementations in the MP handler.

6.2 MP Bytecode Processing Lifecycle

Figure 8 illustrates the subsequent processing pipeline of MP bytecode: Cuttlefish first validates and instruments the bytecode before final JIT compilation to native hardware binary for safe and efficient execution.

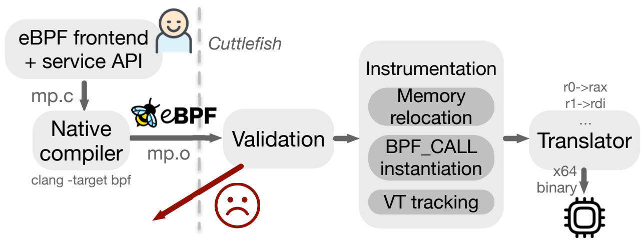


Figure 8 MP bytecode processing workflow.

Validation. Cuttlefish ensures the safety of input bytecode through a validation process similar to that of kernel space eBPF VMs [16, 31]. It rejects programs that attempt memory interactions beyond the allowed indirect KV store access, such as through dynamic memory allocation. Additionally, the use of `BPF_CALL` instructions is restricted to the predefined set of service APIs in Section 6.1, blocking any attempts to invoke unsupported functions through illegal opcodes. Further security checks are described in Section 8.

Note that this step requires that MPs trust the cloud provider and the trading platform operator to not tamper with the ordering of trades or MP code; however, as noted by prior work, this trust is fundamental to a cloud-hosted paradigm [17, 19, 24]. We discuss enforcement mechanisms further in Section 11.

Memory relocation and service API instantiation. As MP programs operate within a constrained memory access environment, Cuttlefish performs memory relocation for those requesting access to KV maps. In particular, it dynamically resolves and replaces symbolic references in the KV map API's `BPF_CALL` instructions with the appropriate memory address during eBPF bytecode loading. This indirection process and the dynamically assigned addresses are invisible and inaccessible to the MPs.

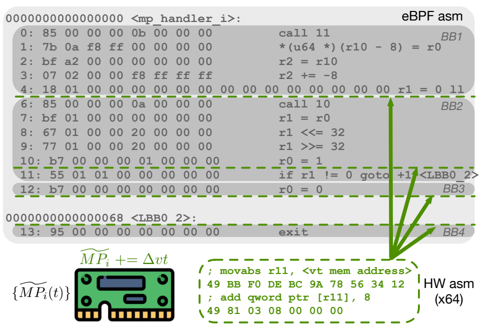
Virtual time tracking instrumentation. To track the virtual time efficiently, Cuttlefish takes a passive, non-intrusive approach via binary rewriting [4, 59], shown in Figure 9:

(1) *Basic block segmentation:* The bytecode is split into basic blocks (BBs)—straight-line sequences of instructions without branches—to facilitate batched virtual time increments Δvt. In addition to `BPF_JMP` call sites, trade submission calls also serve as instrumentation points for capturing the most recent virtual time as the trade needs to be tagged accordingly. For large blocks, Cuttlefish inserts dummy trade submission calls for timely updates of virtual time progress of the MP.

(2) *Virtual time increment instruction:* Cuttlefish emits native machine code (two instructions for x64) at the epilogue of each block to update MP_i's virtual time by addressing the memory location storing $\widetilde{MP_i}$ during JIT translation.

(3) *Offset correction:* The instrumentation also updates the offsets for the JMP instructions. The absence of indirect jumps in the eBPF assembly simplifies this step.

Finally, instead of the slower eBPF interpreter, Cuttlefish adapts eBPF JIT translator to compile IR bytecode into native binaries (e.g., x86_64). This two-tier compilation ensures fair, platform-agnostic virtual time tracking and efficient execution on native hardware.



```
0000000000000000 <mp_handler_i>:
 0: 85 00 00 00 0b 00 00 00         call 11
 1: 7b 0a f8 ff 00 00 00 00         *(u64 *)(r10 - 8) = r0
 2: bf a2 00 00 00 00 00 00         r2 = r10
 3: 07 02 00 00 f8 ff ff ff         r2 += -8
 4: 18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
 6: 85 00 00 00 0a 00 00 00         call 10
 7: bf 01 00 00 00 00 00 00         r1 = r0
 8: 67 01 00 00 20 00 00 00         r1 <<= 32
 9: 77 01 00 00 20 00 00 00         r1 >>= 32
 10: b7 00 00 00 01 00 00 00        r0 = 1
 11: 55 01 01 00 00 00 00 00        if r1 != 0 goto +1 <LBB0_2>
 12: b7 00 00 00 00 00 00 00        r0 = 0

0000000000000068 <LBB0_2>:
 13: 95 00 00 00 00 00 00 00        exit
```

```
; movabs r11, <vt mem address>
49 BB F0 DE BC 9A 78 56 34 12
; add qword ptr [r11], 8
49 81 03 08 00 00 00
```

eBPF asm
BB1
BB2
BB3
BB4
$\widetilde{MP_i} \mathrel{+}= \Delta vt$
$\{\widetilde{MP_i}(t)\}$
HW asm (x64)

Figure 9 Virtual time update instrumentation.

7 Cuttlefish Execution Runtime

Cuttlefish features a practical runtime architecture efficiently implementable on modern clouds. We describe a single VM's execution runtime (Figure 10); extending to multiple is straightforward.

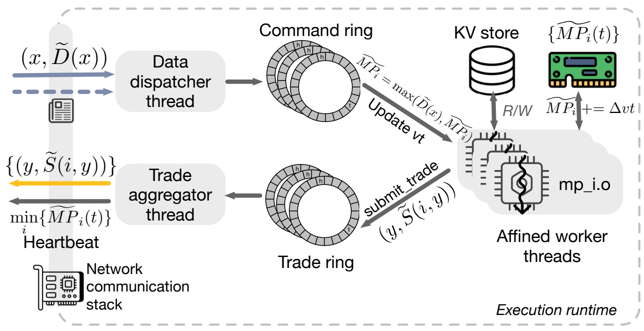
Communication with CES. Cuttlefish runtime interfaces with CES via the data dispatcher and the trade aggregator, both exchanging data streams over a reliable transport layer.

Data dispatcher: The dispatcher manages inbound market data, each coming with an assigned virtual release time.

Trade Aggregator: The trade aggregator gathers tuples from MPs, each comprising a trade decision y and its virtual submission time $\widetilde{S}(i, y)$. These tuples are locally sorted by submission time, batched, and sent to the OB for global sorting. The aggregator is additionally responsible for sending heartbeats to the CES to indicate the latest virtual time reached by all local MPs. The OB uses these heartbeats to decide when it can forward the trade with the lowest virtual submission time in its buffer to the CES safely (i.e., there are no in-flight trades with lower virtual time). This localized handling of sorting and heartbeat calculations enhances the CES's scalability.

Local execution workflow. Cuttlefish's runtime allows consolidating multiple MPs into multi-core VMs for efficiency. It also maximizes CPU utilization (for execution-throughput) and eliminates blocking operations along the data path (for latency). Central to its workflow are the worker threads that execute MPs in parallel, each affined to a dedicated CPU core and configured to run a loaded MP binary. Interaction with the dispatcher and aggregator is streamlined using lock-free, cache-efficient Single-Produce-Single-Consumer (SPSC) rings to minimize processing latency.

The worker threads operate in a busy loop with minimal stalls (e.g., context switching) during the execution. It first polls a batch of command items that contain market data from the command ring. For each market data x processed by MP_i, the worker thread updates the virtual time and invokes the binary with the new market data immediately. Each worker then runs hardware binaries instrumented for uninterrupted virtual time advancement, as outlined in Section 6.2. During execution, MP handlers access the KV store in a thread-safe manner and invoke the redirected function that enqueues trades to the ring with a virtual submission time of the `submit_trade` calls.



$(x, \widetilde{D}(x))$
Data dispatcher thread
Command ring
KV store
$\{\widetilde{MP_i}(t)\}$
$\widetilde{MP_i} = \max(\widetilde{D}(x), \widetilde{MP_i})$
Update vt
R/W
$\widetilde{MP_i} += \Delta vt$
$\{(y, \widetilde{S}(i, y))\}$
Trade aggregator thread
$\min_i\{\widetilde{MP_i}(t)\}$
Heartbeat
submit_trade
$(y, \widetilde{S}(i, y))$
mp_i.o
Trade ring
Affined worker threads
Network communication stack
Execution runtime

Figure 10 Overview of Cuttlefish's execution runtime.

External data handling. In addition to supporting internal data feeds from the CES, Cuttlefish also accommodates external data sources. These interactions are done through a designated gateway (GW) node, as shown in Figure 6, which enforces an additional virtual time latency comparable to that of the public Internet.[6] Just as with current exchanges, we expect that MPs can and will leverage a variety of optimization for these messages, e.g., microwave [7, 6] and satellite [25, 52, 29] networks to surpass wireline c limitations, and/or employ human oversight to tweak parameters in response to changing conditions or unexpected events. Cuttlefish's goal is only to ensure that the post-GW transmission and delivery of the data is fair and predictable. MPs can access the data through the subscribed data context input to the handler, just as they do with normal CES data. We detail the mechanism in Appendix C.

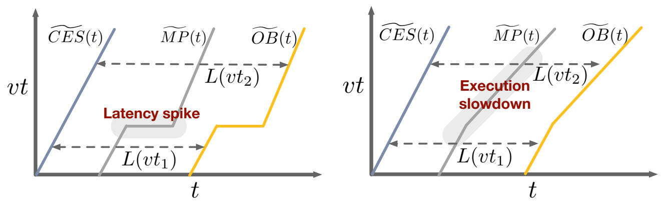
8 Virtual Time Control

So far, we have discussed how the CES broadcasts data and processes aggregated trades from MPs in the order of their virtual submission times at the OB. Another key role of the CES in Cuttlefish is to assign the virtual release time, $\widetilde{D}(x)$, tagged to each market data x for delivery by runtime engines. Intuitively, this virtual time assignment process resembles congestion control but concerns regulating the rate of virtual time progression rather than bytes on a wire.

Objectives. Cuttlefish's virtual time control targets two goals:

1. Minimizing *latency* ($L(vt) = \widetilde{OB}^{-1}(vt) - \widetilde{CES}^{-1}(vt)$), defined as the time difference between the CES releasing market data with virtual time vt ($\widetilde{CES}^{-1}(vt)$) and when the OB hears from all MPs until vt ($\widetilde{OB}^{-1}(vt)$). In simple terms, latency here refers to the minimum time between when market data is produced at the CES and when a trade from an MP using this data can be executed. This definition extends previous end-to-end latency concepts [24] for trades ordered per virtual time.
2. Maximizing *overlay execution-throughput* $\theta = \Delta vt/\Delta t$. This represents the rate of virtual time advancement at the CES or, equivalently, the number of IR instruction cycles available to each MP (per unit of wall-clock time) to process the incoming market data.

[6] A modern commodity switch can provide Tbps capacity [50, 51], which is sufficient to handle typical quantities of external data to exchanges. The GW can scale beyond a single node with a shared clock.



(a) Increased network latency **(b)** Slower execution

Figure 11 Examples when the latency gets impacted by environmental conditions. For the purpose of illustration, the diagram simplifies the discrete steps on events of market data release, invocation, and trade response receipt.

Note that neither of these goals affects correctness, fairness, or predictability. Rather, a good virtual time assignment is important for purely performance reasons. Specifically, virtual time assignment that is too slow can limit the execution-throughput of the exchange, even when the underlying MPs are capable of supporting a higher virtual-time throughput. Conversely, virtual time assignment that is too fast can increase worst-case $L(vt)$ due to MPs that are lagging behind.

Methodology. As Cuttlefish seeks to guarantee fairness across all MPs, stragglers (depicted in Figure 11) can influence virtual time control. Stragglers can arise for two reasons. First, increased network latencies delay market data delivery to MPs, slowing virtual time progression. For example, in Figure 11a, a spike in latency from CES to an MP results in latency growth from $L(vt_1)$ to $L(vt_2)$. In reality, network latencies for both paths (CES-to-MP or MP-to-OB) can affect $L(vt)$. Virtual time assignment should try to mitigate the effect of such spikes. Second, execution slowdowns at an MP (e.g., due to change in processor frequency) can reduce the rate of virtual time progression, cumulatively affecting latencies if these slowdowns are prolonged. Figure 11b illustrates a simplified example of this effect.

To account for them, Cuttlefish presents an easy-to-reason-about approach by coupling its virtual time assignment with real-time evolution. We detail the assignment algorithm in Appendix B. The key idea of our algorithm is similar to that of BBR congestion control protocol [8]. In particular, Cuttlefish measures τ_i, which is the estimated computational capability of MP_i (i.e., how many virtual cycles can the executing engine of MP_i process per unit wall-clock time). We then control the virtual time progression rate based on the bottleneck compute capacity, $\min(\tau_i)$. Similar to BBR, Cuttlefish applies a window cap on the max 'in-flight' virtual time as a guardrail to prevent excessive virtual time assignment in worst-case scenarios.

Failure handling. As described, any MP failure will halt the progress of virtual time. Cuttlefish incorporates timeouts at the CES to detect such events.[7] Because of the determinism of virtual time, with periodic check-pointing of the MP state, the CES can restart and/or relocate such failed components.

We further note that eBPF uses predefined key-value stores for managing state, which makes identifying state and replicating it straightforward. Cuttlefish is also amenable to replication of the MPs themselves. This approach can potentially mask the impact of failures on virtual time progression (§10.4).

[7] Spurious timeouts may degrade performance, but will not affect fairness.

An alternate choice would be to remove the failed MP from the VTC assignment and trade-forwarding logic, and then do a clean restart. The failed MP incurs unfairness in this case. In some abstract sense, Cuttlefish is subject to a CAP-theorem-like limitation: here, the choice is between fairness and progression of virtual time progress in case of failures.

CES failure: CES fault tolerance is beyond the scope of this work; to the best of our knowledge, existing CESes also rely on state replication (e.g., of the order book) for fault tolerance.

Security. While MP code will run alongside other MPs and Cuttlefish components, Cuttlefish benefits greatly from its IR abstraction, basis in virtual time, and validation process (Section 6.2). Potentially problematic instructions or program structures (e.g., attempts to access system memory or cycle counters for creating side channels) are explicitly disallowed. Communication is similarly restricted. As mentioned in Section 5, Cuttlefish adopts a model that forbids MP-to-MP communication similar to prior proposals [24, 19]. However, whereas prior work punts on enforcement, Cuttlefish's constrained runtime provides natural avenues to guarantee it.

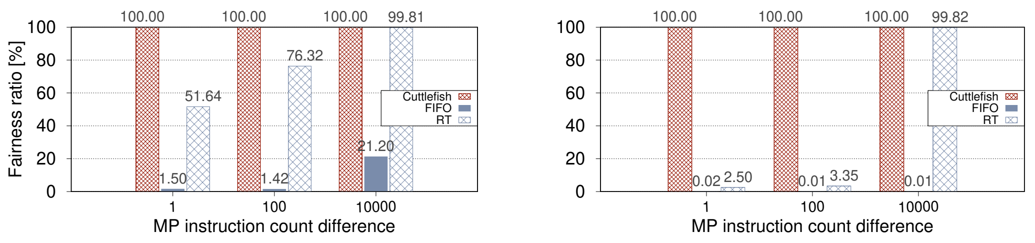
Within the confines of the system, the main potential avenue of attack for MPs is the manipulation of virtual time. Note that MPs with fewer instructions per invocation do not affect system-wide progress, as virtual time advances independently. Instead, MPs might try to deliberately slow their virtual time advancement speed for a chance to influence $\min(\tau_i)$ (if they are the slowest MP in the system) by picking sequences of instructions that have the largest ratio of real-time cost to virtual-time cost. Conversely, if they are already the slowest MP, they could speed up to try to accelerate virtual time advancement. Within Cuttlefish, there is no advantage to manipulating the advancement speed; instead, differences only arise in interactions with external communication and events. We note that the impact is limited: Cuttlefish's GW ensures that external coordination is inefficient (Section 7) and the extent of manipulation is constrained to the gap between the 'slowest' virtual program and the second slowest MP. In the end, however, the critical guarantee that Cuttlefish provides is that any variability in virtual time advancement speed is experienced equally by all MPs.

9 Implementation

To demonstrate Cuttlefish's practicality, we developed and ran a prototype on standard public cloud VMs [44].

Processing MP handler programs. Cuttlefish supports the end-to-end workflow for MP handlers described in Section 6, exposing a single-header interface for Cuttlefish's virtual time services. For verified eBPF bytecodes provided by users, Cuttlefish embeds virtual time tracking transparently through binary writing, while achieving high performance by leveraging the existing eBPF JIT compiler to emit native code [31].

Virtual time cost of instructions. By default, Cuttlefish assigns the virtual time for eBPF instructions based on the equivalent hardware instructions on standard CPU models (e.g., x86_64) and leveraging previous studies that have extensively quantified their costs per machine cycles or reciprocal throughput in modern hardware architecture [14]. For example, `BPF_ADD` is assigned one virtual time unit, with other operations, including handler invocation and `BPF_CALL` instructions for built-in service APIs such as the access to the KV store, scaled accordingly. In Cuttlefish, the relatively small eBPF instruction set [30] simplifies the process.



(a) Same VM **(b)** Different VM types

Figure 12 Cuttlefish guarantees 100% fairness ratio, whereas FIFO and RT-based ordering can only approximate even when MP_b spends non-negligible number of instructions more than MP_a.

While actual instruction cycle times (e.g., memory accesses) can vary in wall-clock time due to microarchitectural effects such as caching, these variances are abstracted away in the virtual time domain. The idea behind such an assignment is to, as much as possible, reduce the dependency of τ on the IR instructions used by the MP in its code. However, exchanges can customize their cost model according to their needs—as long as the models are transparent, the system is fair and predictable.

Supporting efficient execution in virtual time. Cuttlefish employs a reliable, message-based transport [43] for its dispatcher and trade aggregator thread, interfacing with the CES/OB. Worker threads are affined to dedicated cores and are responsible for invoking MP handlers. To facilitate communication between workers and the data dispatcher, Cuttlefish uses a cache-efficient, lock-free Single Producer/Single Consumer circular buffer implementation to instantiate the market and trade rings respectively as detailed in Section 7.

10 Evaluation

10.1 Unfairness in the Cloud

Ordering mechanisms. In addition to Cuttlefish, we examine:

- Response Time (RT) based ordering, which ranks trades by MP processing time. Since CloudEx and DBO either require high-resolution clock synchronization or SmartNIC support—hard to replicate in our environment—we use (1) as a proxy to evaluate behaviors (potential execution unfairness) of CloudEx under perfect network communication fairness and that of DBO's logical clock based on response time. Under identical execution conditions, RT-based ordering should yield 100% fairness ratio.
- FIFO ordering, which processes trades by OB arrival time. Under ideal network and execution conditions, FIFO should also achieve 100% fairness.

Setup. We consider two MPs. MP_b executes N additional primitive IR instructions than MP_a, with all other aspects being identical. In theory, an identical network and execution environment must always prioritize the trades of MP_a over those of MP_b triggered by the same market data. To quantify fairness, we define fairness ratio as the fraction of MP_a trades were (correctly) ordered ahead of corresponding trades from MP_b.

We run experiments on Azure, with one VM operating as CES to generate market data messages at ≈100 µs intervals. We examine two scenarios: (a) MP_a and MP_b run on identical VMs with `Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz`; (b) MP_b uses `Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz` while MP_a running on a different CPU `Intel(R) Xeon(R) CPU`

	Latency (µs)				
	avg.	**p50**	**p90**	**p99**	**p99.9**
MaxRTT	52.04	47.74	49.95	55.85	144.2
Cuttlefish	54.19	50.82	53.49	68.46	166.3

Table 2 $L(vt)$ for 100 MPs: Cuttlefish achieves $\theta = 3279$M vt/s with avg. $\tau_{min} = 3702$M vt/s.

`E5-2673 v4 @ 2.30GHz`. In approach (2), we use lightweight, high-resolution `rdtsc` counters—leveraging constant tsc support across modern processor cores—to measure CPU time at fine granularity and piggyback the measurement with the data for OB ordering. A caveat of this experiment is that the fairness ratio for (1) and (2) can be impacted by the specifics of how the MP algorithm is executed. To reduce this impact, we execute the algorithm using the Cuttlefish runtime environment, which uses strategies like core pinning, CPU isolation, and DPDK to minimize the impact of the OS on execution and network. Further and alternative optimizations (e.g., IPU-based network offload) could further reduce this impact, though other variability would still remain.

Observation. Figure 12a shows the fairness ratios of different ordering mechanisms when two MPs operate on the same type of VMs. In this case, FIFO ordering exhibits significant unfairness as the VM of MP_a experiences a higher RTT compared to that of MP_b. When MP_b executes $N = 10000$ more instructions than MP_a per invocation, the fairness ratio only improves to 78.8%. On the other hand, RT-based ordering shows resilience to network latency disparities. Nevertheless, it incurs about 48% unfairness for $N = 1$, attributable to execution time variances. Despite the mitigation effect of increasing N, RT-based ordering still incurs 0.18% unfairness rate, even at $N = 10000$.

Figure 12b presents a different scenario (b) where MP_a operates on a VM with a slower processor and significantly higher RTT from the CES. FIFO's unfairness remains pronounced at higher N values. RT-based ordering, in turn, experiences a substantially reduced fairness ratio due to the disparity in processor frequencies. E.g., with $N = 1$ or $N = 100$, RT-based ordering's fairness ratio drops below 5%. Throughout these scenarios, Cuttlefish consistently maintains deterministic and 100% fairness ratio. Cuttlefish's fairness guarantee by design remains unaffected by variations in underlying computational power and network latency.

10.2 Performance of Cuttlefish

We evaluate the performance cost of Cuttlefish for fairness, focusing on end-to-end latency $L(vt)$ and execution throughput θ (Section 8). We run Cuttlefish in the Azure cloud environment using `Standard_F16s_v2` instances. Our setup includes a CES VM with a ConnectX-4 NIC and `Intel Xeon Platinum 8272CL CPU @ 2.60GHz`. We use 10 VMs to host 100 MPs and another for the gateway, all in the same VNet/region. The CES broadcasts market data at $\approx$ 100µs intervals.

We also compare the performance of Cuttlefish against its limits (max network latency and minimum compute capability τ_{min} across MPs). To ensure a fair comparison, we measured both network latencies of messages and the computational capabilities of the cores for each MP's core under identical environmental conditions. We record timestamps when market data arrive at the VM (t_1) and when the corresponding trade response leaves the VM (t_2), as well as when they leave (t_0) and arrive at the CES machine (t_3). We then calculate the RTT per VM based on $(t_3 - t_0) - (t_2 - t_1)$ without needing high-resolution clock synchronization.

	Latency (µs)				
	avg.	**p50**	**p90**	**p99**	**p99.9**
MaxRTT	112.0	101.0	113.7	640.3	2984
Cuttlefish	115.5	104.2	116.8	674.5	2996

Table 3 Under higher RTT and background noise: avg. $\tau_{min} = 2488\text{M vt/s}$ and $\theta = 2373\text{M vt/s}$.

Abbreviation	**Algorithm**
bbs	Bollinger Bands Strategy
bmm	Basic Market Making
ema	EMA Mean Reversion
macd	Moving Average Convergence Divergence
macs	Moving Average Crossover Strategy
mmacs	Multiple Moving Average Crossover Strategy
obv	On Balance Volume (OBV) + EMA
psar	Parabolic SAR
rsi	Relative Strength Index
sma	SMA Mean Reversion

Table 4 Example textbook algorithmic trading technical analysis indicators that we have expressed with Cuttlefish's MP handlers.

MaxRTT across VMs is the highest latency across VMs corresponding to the same market data release. Similarly, we measure τ_{min} using `rdtsc` counters.

Table 2 compares Cuttlefish's end-to-end latencies against the MaxRTT across various percentiles. The observed latency discrepancies are attributed to the data flow operations of Cuttlefish, including ring management, batching, and virtual time assignment. Despite trading off some latency for fairness, Cuttlefish shows a commendably low p99.9 tail latency within a typical public cloud setting, in part due to minimal barriers in data release and MP execution. Further, Cuttlefish exhibits a execution-throughput of 3279 M vt/s, which is close to $\tau_{min} = 3702\text{M vt/s}$.[8] In a separate experimental setup, shown in Table 3, we incorporated a VM with a comparatively slower processor at 2.3 GHz frequency and a higher RTT. Cuttlefish adapts well, incurring a modest latency overhead and approximating latency bounds even under stressed conditions.

10.3 Instrumentation Cost

Section 10.2 evaluates the performance of Cuttlefish against latency and throughput limits, showing the end-to-end costs associated with online operations of the virtual time overlay. This subsection investigates the static overhead due to virtual time tracking instrumentation. In particular, we evaluate the impact of this instrumentation on compute capacity.

To quantify the cost, we conducted a stress test under a worst-case scenario: invoking the handler at maximum rate through market data in-memory. Our tests covered various programs shown in Table 4, each with different computational logic and memory access patterns. While these algorithms are simple versions of what is actually used in practice, they demonstrate that, as expected, the Turing completeness of our language is expressive enough to support a wide range of trading algorithms, such as those for statistical arbitrage or directional trading. Interestingly, the interface is also sufficiently intuitive to allow GPT-4 to generate the core algorithmic trading programs that are fully compatible.

[8] This number should not be directly compared with native cycles/s on a superscalar processor [14, 3].

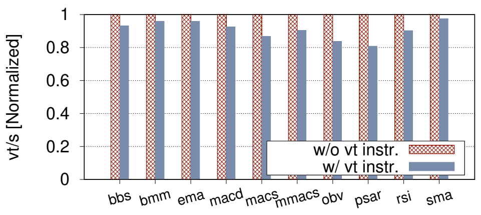


Figure 13 Overhead of virtual time instrumentation across MP handlers in Table 4.

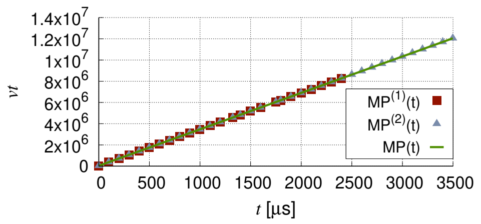


Figure 14 Virtual time series upon a failed MP replica.

As shown in Figure 13, the tracking instructions incur 2–20% overhead in throughput as compared to the vanilla executable, depending on the basic block and branch patterns in the program. In particular, those with thinner loop blocks exhibited a higher virtual time tracking cost. Note the evaluations involve raw algorithms. Cuttlefish can also mitigate the virtual time tracking overhead by providing common computational blocks (often involving intensive loops) as a `BPF_CALL` helper, thus reducing the virtual time tracking breakpoints.

10.4 Exploiting Determinism in Virtual Time For Fault Tolerance

To speed up recovery in the event of failures (§8), Cuttlefish can replicate the execution of an MP handler across different machines by exploiting the *deterministic* virtual time progression across replicas. A physical replica gets integrated as usual with Cuttlefish's virtual time overlay, however, the OB processing the trade that arrives earliest in real-time across the replicas to advance the virtual time for the logical MP. This helps improving the fault-tolerance of Cuttlefish. Figure 14 illustrates a case where Cuttlefish replicates a single MP across 2 different VMs. When a replica $MP^{(1)}$ fails (by killing the worker thread), the virtual time of the logical MP still proceeds as the other replicas $MP^{(2)}$ keeps updating its virtual time. With Cuttlefish, the service provider may trade the cost of replication (and the associated traffic overhead) for lower latency and reliability. Beyond fault tolerance, such replication can also help reduce the latency of Cuttlefish [11].

11 Discussion and Future Work

Cuttlefish explores an admittedly extreme design point, but one that provides strong guarantees, presents an attractive and general interface, and is amenable to efficient implementation. It also happens to fit with existing calls from the financial community and SEC to reduce the barrier to entry for retail traders [36, 6, 7]. The entrenchment of existing exchange architectures means that it is unlikely Cuttlefish will supplant traditional exchanges; however, only time can tell if its tradeoffs allow it to carve out a small, sustainable segment of the market. Even so, as computer scientists, we argue that exploring the challenges and opportunities of this design is worthwhile. This section briefly discuss how Cuttlefish potentially fits into the broader cloud/financial ecosystem.

Heterogenous hardware backends. While Cuttlefish currently focuses on a CPU-based interface and runs on common x64-based cloud VMs, the choice of eBPF IR allows operators to benefit from a broader ecosystem, including mature toolchains that compile eBPF bytecode to ARM64 [31] or offload it to accelerators such as FPGAs [5, 15].

In the extreme case, virtual time is flexible enough to support differentiated instance types and price points. For example, clouds could charge more money for more memory or faster processing (via scaled-up virtual time throughput); they could also sell instances with, for example, 8 CPU cores and 4 FPGAs (by leveraging alternative eBPF backends and the techniques in Appendix A). While this type of support is well out of scope for this paper, one could imagine an IaaS abstraction as rich as traditional cloud offerings, but where each compute device is metered using virtual time.

Interaction with the broader market ecosystem. Securities represent and interact with global assets, and Cuttlefish is only designed to mediate the low-latency algorithmic trading API of the specific exchanges that adopt it; financial activity can occur outside the exchange and even through the slower alternative APIs of an adopting exchange.

In all cases, Cuttlefish eliminates variability and bias between MPs in the exchange. While it potentially also slows them compared to the rest of the world, for securities listed solely at the adopting exchange, Cuttlefish will still be the fastest way to issue buy/sell orders (Section 7). Securities cross-listed at multiple exchanges (Cuttlefish or otherwise) may present opportunities for arbitrage, but that is no different than today, and crucially, the Cuttlefish-enabled cloud platform will never be the determiner of which MP wins the race. As others have noted [48, 24, 7, 19, 40], this prioritization—fairness over pure latency—is a desirable one.

Scalable deployment. Our prototype currently runs with 100 MPs, with the CES broadcasting market data at approximately 100µs intervals. In this setup, the primary bottleneck lies in the CES, which must serialize inbound data ingestion and outbound market data dissemination. One can mitigate bottlenecks with faster processors and high-bandwidth NICs for the CES, which translate to scalability directly. 'Beefier' VM SKUs on the MP side can also help, as Cuttlefish's execution runtime is also designed to scale up with the number of cores, which allows greater MP co-location per machine. To that end, modern cloud providers can offer 'beefy' VM SKUs with over 100 vCPUs [46].

As the number of MPs increases further, components like OB can be scaled out using a distributed architecture: multiple OB instances can be deployed, each responsible for a disjoint subset of RBs (e.g., sharded by trading symbol). These distributed OBs can safely dequeue and batch pending trades, forwarding them to a final merge layer colocated with the matching engine (ME) for order finalization [24].

In parallel, Cuttlefish can benefit from ongoing advances in cloud networking infrastructure. Techniques such as traffic admission control and prioritization [21, 60] can reduce or bound end-to-end latencies. Additionally, scalable multicast engines [26, 48, 18] can help scale CES's market data dissemination, enabling high-rate, low-latency delivery to a larger number of participants.

12 Conclusion

This paper presents Cuttlefish, a fair-by-design execution environment for low-latency algorithmic trading that can run on commercial public clouds. With its virtual time overlay, Cuttlefish abstracts out the variances in the underlying communication and computation hardware, while maintaining low latency and high execution throughput.

References

1 Amazon Web Services. Cryptographic attestation, 2024. `https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html`.

2 Matteo Aquilina, Eric B Budish, and Peter O'Neill. Quantifying the high-frequency trading" arms race": A simple new methodology and estimates. Technical report, Working Paper, 2020.

3 Vignesh Babu and David Nicol. Precise virtual time advancement for network emulation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 175–186, 2020.

4 Vignesh Babu and David Nicol. Temporally synchronized emulation of devices with simulation of networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 1–12, 2022.

5 Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. *Communications of the ACM*, 65(8):92–100, 2022.

6 Eric Budish. High-frequency trading and the design of financial markets, 2023. URL: `https://www.youtube.com/watch?v=OwQjTedWSUM`.

7 Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

8 Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.

9 Gong Chen, Zheng Hu, and Dong Jin. Integrating I/O time to virtual time system for high fidelity container-based network emulation. In Kalyan Perumalla, Margaret Loper, Dong (Kevin) Jin, and Christopher D. Carothers, editors, *SIGSIM-PADS '22: SIGSIM Conference on Principles of Advanced Discrete Simulation, Atlanta, GA, USA, June 8 - 10, 2022*, pages 37–48. ACM, 2022. `doi:10.1145/3518997.3531023`.

10 Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: `http://eprint.iacr.org/2016/086`.

11 Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

12 Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. On studying CPU performance of CloudLab hardware. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

13 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

14 ewasm. Determining wasm gas costs, 2023. `https://github.com/ewasm/design/blob/master/determining_wasm_gas_costs.md`. URL: `https://github.com/ewasm/design/blob/master/determining_wasm_gas_costs.md`.

15 Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

16 Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.

17 Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. CloudEx: A fair-access financial exchange in the cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 96–103, 2021.

18 Junzhi Gong, Yuliang Li, Devdeep Ray, KK Yap, and Nandita Dukkipati. Octopus: A fair packet delivery service. *arXiv preprint arXiv:2401.08126*, 2024.

19 Prateesh Goyal, Ilias Marinos, Eashan Gupta, Chaitanya Bandi, Alan Ross, and Ranveer Chandra. Rethinking cloud-hosted financial exchanges for response time fairness. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 108–114, 2022.

20 James N Gray. Notes on data base operating systems. *Operating systems: An advanced course*, pages 393–481, 2005.

21 Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

22 CME Group. CME group signs 10-year partnership with Google cloud to transform global derivatives markets through cloud adoption, 2019. URL: `https://www.cmegroup.com/media-room/press-releases/2021/11/04/cme_group_signs_10-yearpartnershipwithgooglecloudtotransformglob.html`.

23 Diwaker Gupta, Ken Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: Time-warped network emulation. In Larry L. Peterson and Timothy Roscoe, editors, *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. USENIX, 2006. URL: `http://www.usenix.org/events/nsdi06/tech/gupta.html`.

24 Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. DBO: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 550–563, 2023.

25 Mark Handley. Delay is not an option: Low latency routing in space. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 85–91, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3286062.3286075`.

26 Muhammad Haseeb, Jinkun Geng, Ulysses Butler, Xiyu Hao, Daniel Duclos-Cavalcanti, and Anirudh Sivaraman. Poster: Jasper, a scalable and fair multicast for financial exchanges in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*, pages 36–38, 2024.

27 IEX. The cost of exchange services, 2019. `https://finansdanmark.dk/media/mstbpq23/iex-and-market-data-cost-2019.pdf`.

28 IEX. Updating the IEX exchange architecture for 2021, 2021. `https://www.iex.io/article/updating-the-iex-exchange-architecture-for-2021`.

29 CFA Institute. SpaceX is opening up the next frontier for HFT, 2019. URL: `https://blogs.cfainstitute.org/marketintegrity/2019/06/25/spacex-is-opening-up-the-next-frontier-for-hft/`.

30 Internet Engineering Task Force (IETF). BPF instruction set specification, v1.0, 2023. `https://datatracker.ietf.org/doc/draft-ietf-bpf-isa/`.

31 iovisor. uBPF: User space eBPF vm, 2023. `https://github.com/iovisor/ubpf`.

32 Andrei A. Kirilenko and Andrew W. Lo. Moore's law versus murphy's law: Algorithmic trading and its discontents. *The Journal of Economic Perspectives*, 27(2):51–72, 2013. URL: `http://www.jstor.org/stable/23391690`.

33 Jereme Lamps, David M Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 179–186, 2014.

34 Christian Leber, Benjamin Geib, and Heiner Litz. High frequency trading acceleration using FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322. IEEE, 2011.

35 Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. PrintQueue: Performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

36 Jaime Lizárraga. Increasing competition and improving transparency in U.S. equity markets, 2022. URL: `https://www.sec.gov/news/statement/lizarraga-rule-605-20221214`.

37 John W Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *2012 IEEE 20th annual symposium on high-performance interconnects*, pages 9–16. IEEE, 2012.

38 Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Inf. Control.*, 62(2/3):190–204, 1984. `doi:10.1016/S0019-9958(84)80033-9`.

39 Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018. URL: `https://www.flux.utah.edu/paper/maricq-osdi18`.

40 Vasilios Mavroudis and Hayden Melton. Libra: Fair order-matching for electronic financial exchanges. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 156–168, 2019.

41 Microsoft. B3 partners with microsoft and oracle for systems migration to the cloud, 2022. URL: `https://news.microsoft.com/es-xl/b3-partners-with-microsoft-and-oracle-for-systems-migration-to-the-cloud`.

42 Microsoft. Empowering the future of financial markets with London Stock Exchange Group, 2022. URL: `https://blogs.microsoft.com/blog/2022/12/11/empowering-the-future-of-financial-markets-with-london-stock-exchange-group/`.

43 Microsoft. Machnet, 2023. `https://github.com/microsoft/machnet/tree/main`.

44 Microsoft. Microsoft azure: Cloud computing services, 2023. `https://azure.microsoft.com/en-us/`.

45 Microsoft. Azure virtual network peering, 2024. `https://learn.microsoft.com/en-us/azure/virtual-network/virtual-network-peering-overview`.

46 Microsoft. Sizes for virtual machines in Azure, 2025. Accessed July 2025. URL: `https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview`.

47 Microsoft Azure. Confidential computing, 2024. URL: `https://azure.microsoft.com/en-us/solutions/confidential-compute`.

48 Andy Myers, Brian Nigito, and Nate Foster. Network design considerations for trading systems. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 282–289, 2024.

49 NASDAQ. Nasdaq and aws partner to transform capital markets, 2021. URL: `https://www.nasdaq.com/press-release/nasdaq-and-aws-partner-to-transform-capital-markets-2021-12-01`.

50 Arista Networks. Arista 7132lb datasheet. `https://www.arista.com/assets/data/pdf/Datasheets/7132LB-Datasheet.pdf`.

51 Arista Networks. Arista 7135lb datasheet. `https://www.arista.com/assets/data/pdf/Datasheets/7135LB-Datasheet.pdf`.

52 John Osborne. High-frequency trading over LEO, 2022. URL: `https://josborne.ca/high-frequency-trading-over-leo/`.

53 Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

54 SeaHorn Project. Seahorn: Extending eBPF verification with static analysis, 2019. `https://seahorn.github.io/seahorn/crab/static%20analysis/linux%20extensions/ebpf/2019/07/04/seahorn-ebpf.html`.

55 Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 513–527, 2020.

56 Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems.* distributed-systems.net, 3 edition, 2017.

57 Wikipedia Contributor. Algorithmic trading, 2024. `https://en.wikipedia.org/wiki/Algorithmic_trading`.

58 Liangcheng Yu, Prateesh Goyal, Ilias Marinos, and Vincent Liu. Cuttlefish: A fair, predictable execution environment for cloud-hosted financial exchanges. November 2024. URL: `https://www.microsoft.com/en-us/research/publication/cuttlefish-a-fair-predictable-execution-environment-for-cloud-hosted-financial-exchanges/`.

59 Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert Van Renesse. REM: Resource-efficient mining for blockchains. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1427–1444, 2017.

60 Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 1–18. ACM, 2022.

61 Yuxuan Zhao, Dmitry Duplyakin, Robert Ricci, and Alexandru Uta. Cloud performance variability prediction. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 35–40, 2021.

A Multi-threaded Virtual Time

Cuttlefish can be easily extended to a multi-threaded interface, e.g., to enable the use of heterogeneous hardware or simply to increase the virtual time throughput of the platform.

In parallel execution, each user provides an MP program with multiple handler functions to execute concurrently. The runtime workflow remains the same as described in Section 7, where the dispatcher releases the market data at the same virtual time, preserving the requirements of *R1*. At any virtual time point, a handler function invokes the built-in service API, specifying a target thread ID and the data bytes for inter-thread communication.

If cross-thread coordination is required, we can extend Cuttlefish to deterministic multi-threaded execution by again borrowing inspiration from simulation/emulation parallelization. In particular, we can split thread execution into discrete epochs, each spanning VT virtual time cycles. Threads can schedule inter-thread communication, where data sent during epoch k are batched and released at the start of epoch $k+2$. Thus, before advancing epoch $k+2$, each thread must wait for the others to complete epoch k. This one-epoch offset is to minimize potential latency caused by the loose synchronization barrier among threads. Increasing VT will also reduce the chances of synchronization stalls for faster threads.

While this inter-thread communication introduces a virtual time delay in data forwarding (which mirrors real-time delays in any such communication), it ensures computation fairness as outlined in *R2*. As a result, for a given MP, trade submission times remain deterministic in the virtual time domain, regardless of variations in the runtime execution of individual threads.

B Virtual Time Assignment

CES updates its virtual time $\widetilde{CES}(t)$ when it assigns the virtual delivery time $\widetilde{D}(x)$ for each market data to maintain a stable control loop, accounting for fluctuations in network latency and compute capacity:

$$\widetilde{D}(x) = \min(\widetilde{D}(x-1) + \frac{\min(\tau_i)(G(x) - G(x-1))}{1+\varepsilon}, \widetilde{OB}(t) + \Omega) \qquad (1)$$

where τ_i is the estimated computational capability of MP_i, ε (> 0) is a slack parameter and Ω serves as a cap on the virtual time increase per current $\widetilde{OB}(t)$, where $t = G(x)$.

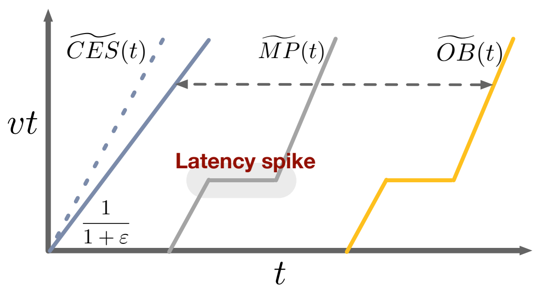


Figure 15 Rate of recovery after a network latency spike is governed by the slack parameter ε.

Measuring τ_i: Cuttlefish periodically profiles τ_i for the executing environment of MP_i. $\min(\tau_i)$ represents the compute capacity of the bottleneck MP. To minimize noise, it uses a low-pass filter such as a moving average.

Role of ε: The idea behind ε is to keep the virtual time assignment rate $(\frac{\widetilde{D}(x)-\widetilde{D}(x-1)}{G(x)-G(x-1)})$ just under the bottleneck compute capacity (τ_i). Upon a network latency spike, the slack can help MPs straggling in virtual time progression to catch up with the CES. The rate of recovery depends on the value of ε. Figure 15 illustrates this effect. Selecting ε involves a trade-off between reducing latency and maintaining high compute-throughput. We use $\varepsilon = 0.05$ in our experiments.

Role of Ω: This cap guard against potential overestimation of $\min(\tau_i)$ In particular, Cuttlefish enforces a cap of $\widetilde{OB}(t) + \Omega$ to bound the maximum amount of 'in-flight' virtual time. The cap is similar to the congestion window cap in network congestion control protocols.

Perception of virtual time assignment for MPs. How does virtual time assignment correlate with MP behaviors? In real-time trading, MPs react to varying market data arrival, striving to minimize processing time based on the latest available information. Similar incentives for rapid response remain in Cuttlefish's virtual time environment.

Unlike traditional trading setups, MPs respond based on virtual time. To facilitate MPs favoring in-time response to market data, Cuttlefish provides access to several virtual time primitives. These include the current virtual time $(\widetilde{MP}_i(t))$, the release time of the triggering market data $(\widetilde{D}(x))$, and a `sleep(vt)` primitive service API to perform NOPs for a specific number of virtual cycles (Section 6.1). This allows MPs to tailor their response strategies, for example, by comparing $\widetilde{MP}_i(t) - \widetilde{D}(x)$ to adjust their computation for the current invocation.

C External Data Handling

As mentioned in Section 7, Cuttlefish ensures that the post-GW transmission and delivery of the data is fair and predictable. Cuttlefish supports two types of external data streams:

Public data feeds. MPs can incorporate external, symmetric data in their algorithms, such as news or market data feeds from other exchanges. Handling these feeds is straightforward: the CES simply integrates the public data stream from the GW with the internal market data when multicasting it to MPs, which subscribe to and deliver the data fairly.

Private event triggers. Cuttlefish's support of private external messages involve two steps:

- *Gateway processing:* The GW sends incoming packets to the corresponding MPs with a per-MP admission rate limit of ρ (to prevent the network from being overwhelmed). It also maintains a loosely synchronized clock with CES and tags each data packet with the admission timestamp t_r, which will be used in the subsequent decision on virtual release time.
- *Dispatcher delivery:* On receiving the private data for MP_i, the data dispatcher buffers and delivers it at the virtual time $\widetilde{CES}(t_r + \Delta t)$. When there is no external data to send, the GW will still periodically send to each runtime engine a heartbeat packet with its t_r. This allows the runtime to release the CES data and advance the virtual time safely by assuring no external private data that might arrive by the scheduled virtual release time. Thus, a larger Δt decreases the chances of introducing additional latencies to the CES data delivery[9].

D Other Related Work

Cloud-hosted exchanges. As detailed in Section 2, others have looked at the fairness of cloud-hosted exchanges. Like other works on cloud-hosted exchanges [17, 24, 18], we seek to retain the liquidity incentives of current markets and do not wish to modify the matching engine. Unlike these approaches, Cuttlefish takes a holistic strategy that not only bridges the gap to a complete guarantee of communication fairness (without assumptions about the trading model or hardware support), but also extends the fairness guarantee to the underlying computation.

Other approaches to fair exchanges. Our goals are explicitly distinct from previous work in the financial community. For instance, Frequent Batch Auctions (FBA) [7] proposes to batch deliver and process market data at a substantially lower frequency of (1 batch/100ms). LIBRA [40] employs a stochastic (non-deterministic) notion of communication fairness for financial exchanges. These strategies modify the existing first-come first-served (FCFS) strategy used by matching engines today; there is a lot of debate on the efficacy and properties of alternative market design choices.

[9] Like ρ, Δt should be also set by the exchange. As the latency of these messages is typically larger overall [28], we configure $\Delta t = 1$ms by default, which is conservative enough for a negligible latency impact on CES data delivery while bounding the relative latency dilation on the external data.