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

Liangcheng Yu[¶,†], *Prateesh Goyal*[¶], *Ilias Marinos*[¶,‡], *Vincent Liu*[†]
[¶]*Microsoft Research,* [‡]*NVIDIA,* [†]*University of Pennsylvania*

Abstract

Recent years have seen rising interest in expanding and equalizing access to low-latency algorithmic trading for a wider group of participants, including retail traders. Cloud-hosted financial exchanges promise a cost-effective platform more accessible to traders. Unfortunately, achieving fairness in cloud environments is challenging due to unpredictable network latencies as well as non-deterministic computation times.

This work presents Cuttlefish, a fair-by-design algorithmic trading platform that can run in cloud environments. The idea behind Cuttlefish is the efficient and robust mapping of real operations to a novel formulation of 'virtual time'. With it, Cuttlefish pushes fairness to the extreme regardless of the underlying network communication and computation hardware. Our implementation and evaluation not only validates the practicality of Cuttlefish, but also shows its operational efficiency on public cloud platforms.

1 Introduction

Low-latency algorithmic trading is responsible for much of the efficiency of modern financial markets, promoting accurate/timely pricing of securities, higher liquidity, and lower trade costs for all investors—institutional or retail [31]. The goal of low-latency algorithmic trading is to process incoming market data and issue trades as quickly as possible to take advantage of ephemeral market-making and arbitrage opportunities [54]. Compared to the broader set of algorithmic trading strategies, which can involve longer time scales and even humans in the loop, the emphasis of low-latency approaches is on reaction time rather than the complexity/dimensionality of the algorithm. A sizable fraction of activity in today's exchanges is the result of this class of trades [4, 24].

It is, thus, worth examining, in depth, the viability of this type of trading in the context of the recent surge of interest in cloud-hosted financial exchanges, e.g., from major exchanges like NASDAQ, CME, LSE, and B3, as well as cloud providers like Microsoft, Amazon, and Google [22, 41, 42, 46]. Exchanges are interested in moving to the cloud for a variety of reasons, including better scaling, fewer outages, improved cost savings, and a potentially broader customer base as, theoretically, any user can rent a machine in the same region as the Central Exchange Server (CES) and participate without the logistical hurdles involved in installing and maintaining on-premise hardware [17–19, 24]. 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.

The primary challenge in this migration is that the cloud platforms are not nearly as fair/predictable as current exchange infrastructure, where exchanges deploy layer-1 switches and equal-length network cables to eliminate differences in propagation delay [17, 18, 24] and where Market Participants (MPs)[1] implement custom operating systems and network stacks on top of bare metal to control differences and compete against other participants in compute performance [9, 24, 33, 36, 38]. As the order in which the exchange server processes competing trades governs the outcome of the competition, there are considerable advantages to winning that race, and cloud practicalities (e.g., a noisy neighbor, thermal variations, or unscheduled cloud-provider maintenance operations) [39, 50, 52], can significantly affect its outcome.

Our goal in this work is a fair, predictable execution environment for cloud-hosted financial exchanges, where the outcome of races are based on the strength of the MPs' designs rather than (un)lucky performance fluctuations of their underlying computation and communication infrastructure. In addition, the system should provide low-latency and high throughput.

We note that recent work in cloud exchanges has made large strides in taming variability in communication infrastructure [17, 19, 24]. However, despite their progress, they continue to struggle to provide complete guarantees on the communication latency of market data and trade delivery. CloudEx [17], for instance, breaks down when the clock synchronization error gets too high [37] or when network latency spikes beyond a preconfigured threshold. DBO [19, 24] addresses some of these

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

issues with its use of logical clocks but breaks in the presence of non-speed-race trades (i.e., those that are based on an aggregation of multiple market data points) and when there is variation in latency between the release buffer (RB) and the MP machine.

In this paper, we present Cuttlefish[2], the first cloud-hosted exchange for low-latency algorithmic trading that ensures fair, predictable execution in the face of compute and communication variance. Not only is Cuttlefish the first to address differences in compute performance in these environments, *its focus on both dimensions helps it to achieve more complete guarantees of communication fairness even when compared to prior work that specializes in communication.*

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 [14, 20, 24, 53]. Further, beyond just the fairness of computation and communication, the system needs to have low latency and high throughput to be usable by MP and their trading algorithms, where latency is defined as the minimum time between production of market data at the CES and the execution of trade from an MP using this data (see §7), and throughput is defined as the amount of calculations that MPs can perform per unit time.

Cuttlefish achieves the above using an efficient and robust mapping of real operations to 'virtual time' from a platform-agnostic intermediate representation (IR). 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 computation processes and guarantees fairness, regardless of the varying latencies in the communication of market data and trade responses or variations in computation platforms.

This concept of virtual time mirrors that of other applications such as co-simulation [5, 6, 11, 32], which coordinates virtual time for concurrent emulation and simulation processes. Cuttlefish takes a step further by extending the concept to real-time systems and developing an end-to-end trading platform. Cuttlefish's architecture is, thus, a combination of (1) a platform-agnostic 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.

Our prototype not only demonstrates the viability and practicality of Cuttlefish's design, but also the efficiency in its exchange operations while being deployable on commercial cloud virtual machines. To summarize, this paper makes the following contributions:

- We propose Cuttlefish, the first cloud exchange platform to tackle compute fairness and, as a result, simultaneously address persistent gaps in the communication fairness of existing systems.
- We introduce a precise virtual time overlay and mapping strategy to abstract the differences in the underlying data communication and computation hardware while maintaining low latency and high compute throughput to MPs.
- Finally, we evaluate Cuttlefish using an end-to-end implementation on a real cloud platform. When serving 100 MPs, Cuttlefish not only guarantees fairness in communication and computation, but also low end-to-end latencies ($<$ 150µs p99.9 tail latencies) and compute throughput close to their limits. We will open-source the code upon publication.

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

2 Background and Goals

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. This shift from human-driven to computer-driven trading has opened up the possibility of so-called algorithmic trading techniques, now the cornerstone of modern financial markets.

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 [4, 24, 31, 35, 54]. 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 [33, 36].

Exchanges that support low-latency algorithmic trading expose a simple top-level abstraction: (1) the exchange broadcasts market data to all MPs, delivered at time $\{D\}$, (2) the MPs analyze the data, and (3) they send trades back to the exchange at time $\{S\}$, processed in the order of submission. MPs can also integrate external data into their strategies, but the import of these data is orders of magnitude less frequent than the data processing of the core market.

Exchanges also provide alternative trading interfaces, but they are typically three to four orders of magnitude higher latency and used by MPs for less latency-critical trading (e.g., quantitative trading). Trades made through these alternate interfaces eventually become part of the incoming market data [1]. The design of these alternate trading interfaces is orthogonal to this work.

Cloud-hosted exchanges. As mentioned, there has been sig-

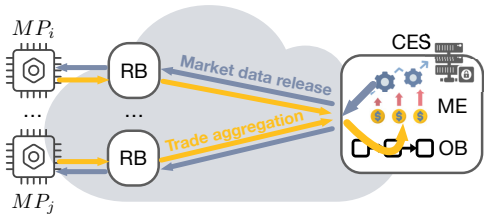


Figure 1: Basic structure of cloud-hosted exchanges. All systems components are controlled by the trusted cloud provider.

nificant recent interest in cloud-hosted exchanges for reasons including better scaling, fewer outages, improved cost savings, and broader access to the financial markets [17, 19, 24, 35].

Figure 1 depicts the main components of these exchanges. At the core of these systems is the Central Exchange Server (CES), which disseminates market data to all MPs through Release Buffers (RBs) or equivalents. 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. There, trades are first enqueued and sorted at the Ordering Buffer (OB) and then processed in FIFO order by the Matching Engine (ME). The ME finally updates the limit order book and generates a new batch of market data.

Although moving to a cloud-hosted solution requires MPs to trust the cloud provider and trading platform to not steal or manipulate the MP trading code and execution, such requirements are typically enforced through contracts and regulatory bodies such as the SEC and privacy laws (similar to the existing restrictions on tampering in on-premise platforms). Recent advances in hardware (e.g., Intel SGX [12]) as well as efforts in cloud confidential computing [3, 45] could provide additional guarantees of confidentiality and security.

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 computation and communication, performance variations can quickly overwhelm MPs' careful optimizations, creating a world where MPs win and lose not on the strength of their algorithms but on purely external factors (e.g., noisy neighbors or provider monitoring/maintenance).

To illustrate, consider the simple scenario of delivering market data x for MP_i and MP_j in Figure 2a. 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 SKUs), and the same software stack (OS and the MP's trading algorithm), for the same algorithm, the computation time can still vary significantly (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 2b. In both cases, any disparity, even at nanosecond time scales, can advantage/disadvantage an MP.

Performance bias and variability can occur for any number of reasons, including everything from non-deterministic software operations to machine-specific hardware wear and thermal effects, with some studies citing CoVs of performance in bare-metal infrastructure of up to 30% [39, 52]. These are on top of noisy neighbor and hypervisor effects introduced in cloud deployments.

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 when MP_i submits trade y.
$\widetilde{D}(x)$	Virtual time assigned to deliver data x.
$\widetilde{S}(i,y)$	Virtual time when MP_i submits trade y.
$\widetilde{CES}(t)$	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.

While it may be possible for cloud providers to try to tame this effect, e.g., preventing all multi-tenancy, removing all management/telemetry infrastructure, or carefully controlling temperature and wear, doing so would substantially cut into the cost and scalability advantages of cloud-hosted exchanges. In this work, we assume a traditional cloud execution model where the provider is responsible for compute and network infrastructure and the infrastructure is fundamentally variable; alternative setups are out of the scope of this work.

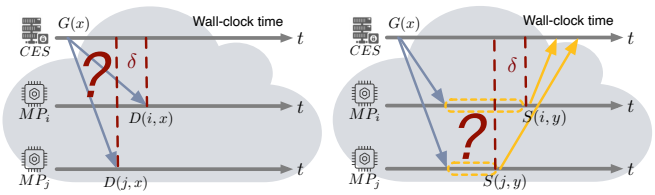
Our goal. Our goal in this work is to create a cloud-based trading platform that is fair and predictable—where the outcome of competitions is based on only the properties of MPs' algorithms rather than luck. More specifically, we target:

R1 *Communication fairness:* (a) All MPs should get access to the market data points at the same time ($\{D\}$) 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 *Computation fairness:* For any trading algorithm permissible on the platform, A, *given* any execution of the algorithm in the platform, the submission time of the generated trades, $\{S_A\}$, is completely defined by the delivery time of the input data to the algorithm $\{D_A\}$. Consequently, given *R1*, for a given trading algorithm, the computation time to generate the trades should not vary across MPs.

In addition, MPs should experience low latency and high compute throughput for their trades and algorithms. For the coarse-grained external data, Cuttlefish also ensures its fair and predictable delivery (see §6).

Recent work on fair cloud exchanges. We note that prior work [17, 19, 24] has already begun to explore the design of cloud exchanges but with a singular focus on communication. Unfortunately, even for *R1*, their guarantees are incomplete.



(a) Communication unfairness (b) Computation unfairness

Figure 2: Simultaneous data delivery and compute fairness are difficult, even under the same market data, MP algorithm, and execution platform.

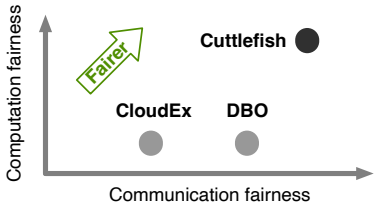


Figure 3: Recent cloud-hosted exchange proposals [17–19, 24] target only communication fairness.

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—likely a strong assumption in distributed systems [37]—the guarantees break whenever there is a latency spike exceeding the threshold. Such latency spikes can occur unpredictably in cloud environments [19, 24, 34], violating *R1*. Configuring conservative threshold values can help (at the cost of higher latency penalties) but are not a complete remedy [14, 19, 24].

More recent work, DBO [19, 24], 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 (an illusion) of *R1*. DBO's guarantees are tailored for a specific trading pattern, namely trigger-point-based high-speed trades. Trades that do not fit this model (e.g., slower trades or trades triggered using two or more data points) are not necessarily fair. DBO also has an assumption of equal latency between MPs and their local RBs, which presents additional challenges for practical deployments in cloud environments.

3 Virtual Time in Cuttlefish

Cuttlefish tackles both computation and communication fairness simultaneously. We find that including compute fairness not only serves to present a more predictable execution platform—as cloud compute can be just as variable as cloud networking—it also naturally addresses the fundamental limitations of existing work on the communication side.

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 that is independent of wall-clock time.

Of particular relevance is the use of virtual time in high-fidelity emulation of processes interacting over a network [5, 6, 11, 23, 32]. 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 real 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 of pausing and resuming processes to ensure that all processes are synchronized and events are sequenced correctly according to their virtual time.

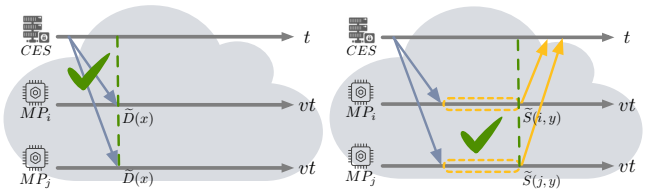
Network emulation vs. low-latency algorithmic trading. Cuttlefish takes an analogous approach by assigning virtual time to all communication and computation—down to an instruction level. Like emulation, Cuttlefish benefits from the ability to control the fine-grained progression of virtual time for each 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 to faster ones.

Cuttlefish virtual time, illustrated. Figure 4 depicts the operation of virtual time in Cuttlefish. For simplicity, we will ignore component failures but discuss how Cuttlefish can be extended to handle them in §7.

The Cuttlefish CES operates unrestricted in *wall-clock time*, while the MPs track and adhere to *virtual time*. The notation used in the figure and the remainder of the paper are summarized in Table 1.



(a) Communication fairness **(b) Computation fairness**

Figure 4: Computation and communication fairness are achievable in the virtual time domain through the deterministic control of virtual time passage and the quantification of computation using virtual time.

1. [Figure 4a] For fair market data delivery (*R1a*), the CES picks 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 4b] For compute fairness (*R2*), Cuttlefish provides a deterministic accounting of computation—specifically, the instruction cycle count executed by its platform-agnostic virtual machine substrate.
3. [Figure 4b] For fair trade ordering (*R1b*), trades from an MP are marked with the virtual time at which they are generated, $\widetilde{S}(i,y)$ and $\widetilde{S}(j,y)$. Similar to DBO, Cuttlefish features an ordering buffer that forwards these trades to the CES based on their generation time.

Strict adherence to virtual time on all MPs ensures both communication and computation fairness. The CES's ability to bridge wall-clock and virtual time ensures a strong relationship between the two (see §7).

4 Design Overview

Building a practical, end-to-end system around the above virtual time abstraction presents a range of technical challenges: How can we instantiate virtual time and track its advancement in a real system? How can we minimize overheads while ensuring fairness? What should the system do when there are spikes in network latency or a slowdown in a particular MP's progress? We discuss them in the remainder of this paper.

Workflow. Figure 5 depicts the high-level architecture of Cuttlefish. On top of the classic architecture of cloud-hosted exchanges, Cuttlefish adds three primary components to instantiate the virtual time overlay:

❶*MP algorithm representation via eBPF VM bytecode [§5]:* To account for the amount of computation deterministically, Cuttlefish leverages a platform-agnostic IR that is based on the eBPF Virtual Machine (VM) 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.

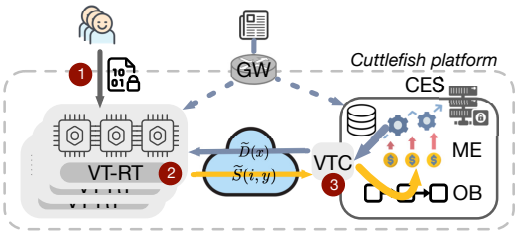


Figure 5: Overview of the Cuttlefish platform.

❷*Virtual time execution runtime (VT-RT) [§6]:* 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) [§7]:* 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.

We note that our virtual time abstraction is naturally compatible with external data events. In particular, we can take external data events and deliver them to MPs at any desired virtual time. Cuttlefish decides the delivery time of external data events to preserve competition in bringing external data to the trading machine faster than other MPs (see §6).

The cost of fairness. In exchange for fairness/predictability, Cuttlefish incurs modest overheads on MP execution. Some of this is due to the extra instrumentation to track and control virtual time, which incurs a relatively small overhead of ∼2–20%.

More fundamentally, guaranteeing that all MPs have an equal opportunity to respond to market data 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 compute.

In the end, however, in our evaluation on a public cloud and 100 MPs, Cuttlefish was able to achieve high performance in the presence of real-world performance variation (see §9.3). If higher throughput is needed, better hardware or multi-threaded execution (Appendix A) can help. Regardless, Cuttlefish still guarantees fairness and predictability in all cases.

5 MP Algorithm Representation

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

```
#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 6: An example MP pseudocode in high-level language using Cuttlefish service APIs, which includes a narrow interface to a KV store for stateful invocations.

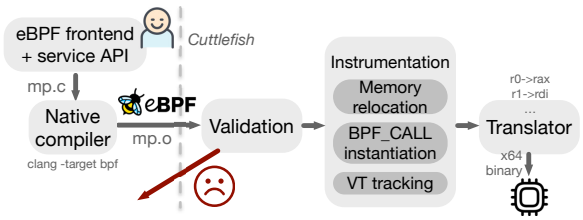


Figure 7: MP bytecode processing workflow.

5.1 Programming Interface

Recall from §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 6 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 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 §8).

eBPF IR. Although any platform-agnostic runtime could serve as a virtual hardware substrate, Cuttlefish chooses an IR based on the eBPF Virtual Machine (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) [7, 30]. More importantly, the simple eBPF 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 [16, 30, 51].

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, Cuttlefish provides a Turing-complete interface [16, 51] for MPs to implement trading strategies, and we show several 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.

5.2 MP Bytecode Processing Lifecycle

Once the MP provides their bytecode to Cuttlefish, Figure 7 illustrates the subsequent processing pipeline: Cuttlefish first validates and instruments the bytecode before final JIT compilation to native hardware binary for safe and efficient execution.

Validation. Cuttlefish ensures the safety of input bytecode through a validation process similar to that of kernel space eBPF VMs [16, 30]. 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 §5.1, blocking any attempts to invoke unsupported functions through illegal opcodes. Further security checks are described in §7.

This step, similar to previous work in cloud exchanges, requires that MPs must trust the cloud provider and the trading platform operator to not tamper with the ordering of trades or MP code. This is usually enforced through contracts, auditing bodies, and regulatory laws. As mentioned in §2, recent

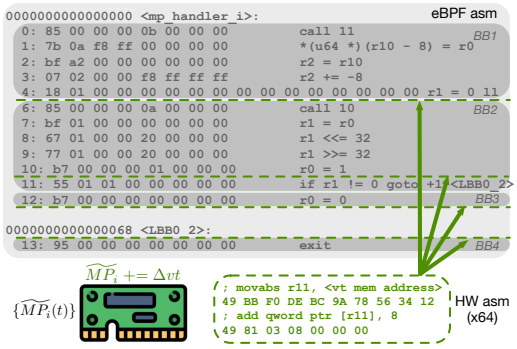


Figure 8: Virtual time update instrumentation.

advances in cloud confidential computing [3, 12, 26, 45] can also benefit Cuttlefish by providing cryptographic attestation and secure enclaves to isolate sensitive data and MP code during both instrumentation and execution.

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 (similar to [6], shown in Figure 8):

(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 executing bytecode with the slower eBPF interpreter, Cuttlefish adapts the eBPF JIT translator to convert it into native machine binaries, such as x64. This decoupled two-tier compilation strategy allows for tracking virtual time fairly in a platform-agnostic manner while executing efficiently on the native hardware target without interruption.

Remark: We emphasize that, while we focus on a CPU-based interface, this does not preclude future extensions to alternative backends, nor are the features discussed intended to be exhaustive. Rather, they are the most relevant to meet the demand of today's low-latency algorithmic trading on the cloud. In fact, our approach for virtual time tracking is platform-agnostic and the choice of eBPF IR allows operators to benefit from a broad ecosystem, including support for offloading eBPF program execution to accelerators such as FPGAs [7, 15].

6 Cuttlefish Execution Runtime

A critical component of Cuttlefish is a generic runtime platform that can execute trading operations efficiently while enforcing fairness. To ensure that the runtime architecture is performant, practical, and efficient to implement, Cuttlefish consolidates multiple MPs into multi-core VMs. In this section we focus on a single VM's execution runtime (depicted in Figure 9); extension to multiple runtimes is straightforward.

Communication with CES. Cuttlefish's runtime engines interface with the CES through two interfaces: the data dispatcher and the trade aggregator, both of which exchange data streams using a reliable transport layer.

Data dispatcher: The dispatcher manages inbound market data, each of which comes 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 is 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 is designed to maximize CPU utilization (for compute-throughput) and eliminate 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 §5.2. During execution, MP handlers access the KV store in a thread-safe manner and invoke the redirected function that

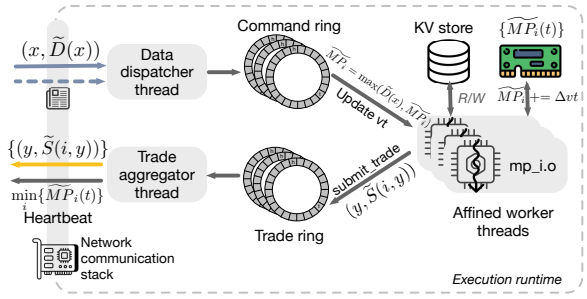


Figure 9: Overview of Cuttlefish's execution runtime.

enqueues trades to the ring with the virtual submission time at the point of `submit_trade` calls.

External data handling. In addition to supporting internal data feeds from the CES, Cuttlefish also accommodates MPs that desire the ability to react to external data sources. All such interaction is done through a single gateway (GW) node[3], as shown in Figure 5. Clearly, Cuttlefish has no control over the computation or communication of nodes outside of the cloud provider's purview. Just as with current exchanges, we expect that MPs can and will leverage heterogeneous hardware, e.g., microwave [8, 9] and satellite [25, 28, 49] networks to surpass wireline c limitations, and/or employ human oversight to tweak parameters in response to changing conditions or unexpected events. Rather, Cuttlefish's goal with external data is to ensure that the post-GW transmission and delivery of the data is fair and predictable. There are two main 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. For such public data, all MPs in Cuttlefish have equal access to the same external events at the same release times. Handling a symmetric data feed is straightforward: the CES simply integrates the public data stream from the GW with the internal market data when multicasting it to MPs, which will all deliver the data at the same virtual time.

Private event triggers: Cuttlefish also supports MPs receiving private external messages. Handling this data involves two steps:

1. *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.
2. *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[4].

Together, they provide deterministic post-GW delivery of external private data in the virtual time domain. MPs can then access the data through the subscribed data context input to the handler, just as they do with normal CES data.

7 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.

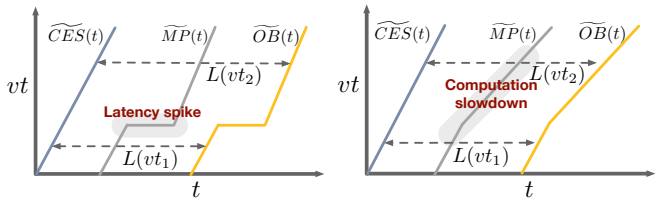
Objectives. Cuttlefish's virtual time control targets two goals:

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 compute-throughput* $\theta = \Delta vt / \Delta t$. This represents the rate of virtual time advancement at the CES or, equivalently, the number of eBPF IR instruction cycles available to each MP (per unit of wall-clock time) to process the incoming market data.

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 compute-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.

[3] A modern commodity switch can provide Tbps capacity [47, 48], which is sufficient to handle typical quantities of external data to exchanges. Cuttlefish can scale beyond a single GW with a shared clock.

[4] Like ρ, Δt should be also set by the exchange. Since these streams typically traverse the Internet, the network latency is significantly higher ($O(\text{ms})$) than the main CES data stream ($O(\mu\text{s})$) [27]. We configure $\Delta t = 1$ms proportionally to Internet latency, conservative enough for a negligible latency impact on CES data delivery while bounding the relative latency dilation on the external data.



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

Figure 10: 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.

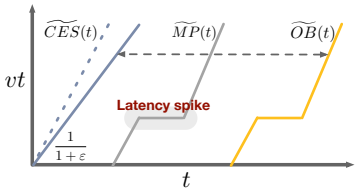


Figure 11: The rate of recovery after a network latency spike is governed by the slack parameter ε.

Stragglers. As Cuttlefish seeks to guarantee fairness across all MPs, straggers (depicted in Figure 10) can influence virtual time control. Stragglers can arise for a couple of reasons.

First, increased network latencies lead to delayed market data delivery to MPs, slowing virtual time progression. For example, in Figure 10a, 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, computation 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 10b illustrates a simplified example of this effect.

Assignment algorithm. Cuttlefish presents an easy-to-reason-about approach by coupling its virtual time assignment with real-time evolution. The strategy is similar to the BBR congestion control protocol [10] with a virtual time rate control based on the estimated bottleneck capacity and a window cap on in-flight requests.

Virtual delivery time computation: 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:

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

where τ_i is the estimated computational capability of MP_i (how many virtual cycles can the executing engine of MP_i process per unit wall-clock time), ε (>0) is a slack parameter and Ω serves as a cap on the virtual time increase per current $\widetilde{OB}(t)$, where $t = G(x)$.

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 11 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 serves as a guardrail to prevent excessive virtual time assignment in worst-case scenarios, e.g., due to potential overestimation in $\min(\tau_i)$. To guard against such mis-estimations, 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 (§5.1). 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.

Failures. As described, any component failure will halt the progress of virtual time. Cuttlefish can be extended to recover from MP failures. Timeouts can be incorporated at the CES to detect such events[5]. 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 (§9.4).

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

[5]Spurious timeouts may degrade performance, but will not affect the fairness of Cuttlefish.

failures. 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 constrained memory model, basis in virtual time, and validation process (§5.2). Any instructions or program structures that could be problematic (e.g., attempts to access system memory or cycle counters for creating side channels) are disallowed in Cuttlefish.

The more subtle opportunity for misbehavior involves MPs deliberately slowing their virtual time advancement speed to potentially influence $\min(\tau_i)$ by picking sequences of instructions that have the largest ratio of real-time cost to virtual-time cost. Cuttlefish thwarts this manipulation by periodically measuring the τ_i of each Cuttlefish execution runtime using an independent profiling codelet. This standard reference program is public so all MPs are on equal, predictable footing.

8 Implementation

To demonstrate Cuttlefish's practicality, we developed a prototype runnable on standard VMs of public clouds [44].

Processing MP handler programs. Cuttlefish supports the end-to-end processing workflow for MP handlers described in §5. It provides a single-header inclusion of interfaces for services in Cuttlefish's virtual time environment. 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 [30].

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., x64) and leveraging previous studies that have extensively quantified their costs per machine cycles or reciprocal throughput in modern hardware architecture [2]. 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 [29] simplifies the process. 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 §6.

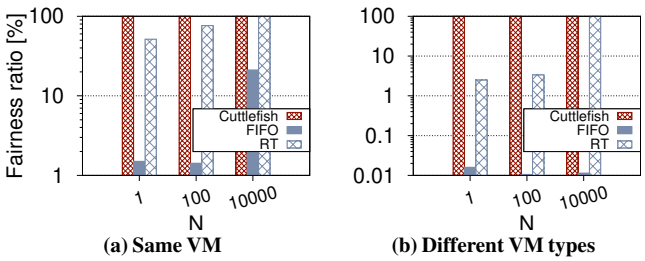


Figure 12: Cuttlefish guarantees 100% fairness ratio, whereas FIFO and RT-based ordering can only approximate even when MP_b spends non-negligible number of instructions than MP_a. For instance, under $N = 10000$ and the same VM, RT-based ordering still incurs 0.18% unfairness rate.

9 Evaluation

Our evaluation focuses on the following questions:

- Does Cuttlefish's fairness make a difference? (§9.1)
- What is the achievable latency and throughput, and how do they compare to optimum? (§9.2)
- How is the instrumentation 'tax' associated with virtual time tracking? (§9.3)
- How does Cuttlefish's predictability help? (§9.4)

9.1 Unfairness in Cloud

We consider a simple experiment to show how cloud environments can cause network and compute unfairness.

Set up. We study a basic scenario of two MPs, MP_a and MP_b. MP_b executes N additional primitive IR instructions (`BPF_ADD`) in the handler's epilogue compared to MP_a, with all other aspects being identical. In theory, an identical network and compute environment must always prioritize the trades of MP_a over those of MP_b triggered by the same market data. To quantify fairness in this experiment, we define a fairness ratio, representing what fraction of MP_a trades were ordered ahead of corresponding trades from MP_b.

Ordering mechanisms. We additionally examine two strategies: (1) Response Time (RT) based ordering, which ranks trades based on the measured processing time for an MP. 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 compute unfairness) of CloudEx under perfect network communication fairness as well as that of DBO's logical clock based on response time. Under identical compute conditions, RT-based ordering should have a fairness ratio of 100%. (2)

	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: $L(vt)$ when serving 100 MPs with 11 VMs. Cuttlefish's θ achieves 3279M vt/s with avg. τ_{min} 3702 M vt/s.

	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: Scenario with a higher RTT and background noise. Avg. $\tau_{min} = 2488$M vt/s and $\theta = 2373$M vt/s.

FIFO ordering, which processes trades based on their arrival at the OB. In FIFO ordering, under an ideal network and compute environment, the fairness ratio should be 100%.

We run experiments on a public cloud, with one VM operating as CES to generate market data messages at $\approx$100 μs intervals. We investigate two distinct scenarios: (a) where MP_a and MP_b run on same types of VMs with processors `Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz`; (b) where MP_b using `Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz` while MP_a running on a different processor `Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz`. For approach (2), we measure the response time at fine granularity by calculating the CPU time based on `rdtsc` counters and piggyback the measurement with the market data for ordering at OB. 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 and DPDK to minimize (but not eliminate) the impact of OS on computation and network.

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 computational time variances. Despite the mitigation effect of increasing N, RT-based ordering doesn't achieve a fairness ratio of 100%, 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. For example, 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.

9.2 Performance of Cuttlefish

To evaluate the performance costs of Cuttlefish for fairness, particularly in how its end-to-end latency $L(vt)$ and compute throughput θ (as outlined in §7). We run Cuttlefish in a standardized cloud environment using `Standard_F16s_v2` instances. Our setup includes a VM running as the CES, with a ConnectX-4 NIC featuring 25Gbps ports and a `Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz`. We utilize 10 VMs of the same type in the same region to host 100 MPs[6] and another one for the gateway. In this setup, the CES broadcasts market data messages to all MPs at $\approx 100\mu s$ intervals.

We also compare the performance of Cuttlefish against its limits (max network latency and minimum compute capability 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. MaxRTT across VMs is the highest latency across VMs corresponding to the same market data release. Similarly, we measure τ_{min} using the lightweight, high-resolution `rdtsc` counters that leverage the common support for constant tsc across all cores in modern processors.

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 public cloud setting, in part due to minimal barriers in data release and MP execution. Further, Cuttlefish exhibits a high compute-throughput of 3279 M vt/s[7]. To our knowledge, low-latency trading algorithms used by HFTs are typically simple and optimized for speed. We believe, that the throughput offered by Cuttlefish should be enough to implement such algorithms.

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 displayed adaptability to this change, effectively approximating the

[6] Approximately the maximum scale of most existing exchanges [1, 24].

[7] This number should not be directly compared with native cycles/s on a superscalar processor [2, 5].

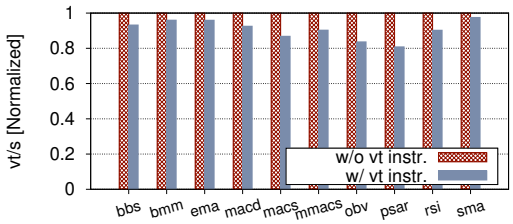


Figure 13: Overhead of virtual time instrumentation across a range of MP handlers in Table 4.

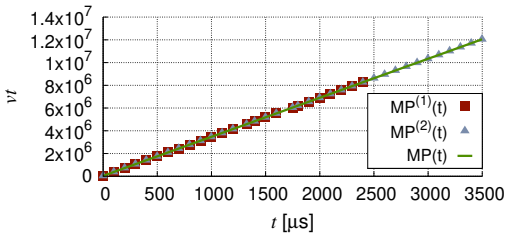


Figure 14: Virtual time series upon a failed MP replicas.

latency bound even under stressed conditions.

9.3 Instrumentation Cost

§9.2 evaluates the performance of Cuttlefish against the 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 the additional instructions for virtual time tracking. In particular, we evaluate the impact of such 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 logic and key-value store access patterns. We found that Cuttlefish's handler interface 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.

As shown in Figure 13, the tracking instructions incur 2–20% overhead in throughput as compared to on 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 number of virtual time tracking breakpoints.

9.4 Determinism for Fault Tolerance

To speed up recovery in the event of failures (§7), 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 9.4 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 [13].

10 Related Work

Cloud-hosted exchanges. Like other works on cloud-hosted exchanges [17,18,24], we seek to retain the liquidity incentives of current markets and do not wish to modify the matching engine. Unlike these approaches (as detailed in §2), 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) [9] 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.

Improved cloud network. Traffic admission control and prioritization [21, 55] can lower (or even bound) network latencies benefiting Cuttlefish and other strategies. Recent advances in cloud confidential computing [3, 12, 45] also help provide stronger guarantees of confidentiality and security for cloud-hosted exchanges [26].

11 Conclusion

This work presents Cuttlefish, a fair-by-design low-latency algorithmic trading platform that can run on uncertain cloud-like environments. By introducing its virtual time overlay, Cuttlefish abstracts out the variances in the underlying communication and computation hardware, while maintaining low latency and high compute throughput.

References

[1] The Cost Of Exchange Services. https://finansdanmark.dk/media/mstbpq23/iex-and-market-data-cost-2019.pdf.

[2] Determining wasm gas costs. https://github.com/ewasm/design/blob/master/determining_wasm_gas_costs.md, 2023.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Eric Budish. High-frequency trading and the design of financial markets, 2023.

[9] 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.

[10] 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.

[11] 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.

[12] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

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

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

[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.

[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.

[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.

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

[28] CFA Institute. Spacex is opening up the next frontier for hft, 2019.

[29] Internet Engineering Task Force (IETF). BPF Instruction Set Specification, v1.0. https://datatracker.ietf.org/doc/draft-ietf-bpf-isa/, 2023.

[30] iovisor. ubpf: User space ebpf vm. https://github.com/iovisor/ubpf, 2023.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Jaime Lizárraga. Increasing competition and improving transparency in u.s. equity markets, 2022.

[36] 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.

[37] Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Inf. Control.*, 62(2/3):190–204, 1984.

[38] Donald MacKenzie. How fragile is competition in high-frequency trading. *Tabbforum, March*, 26, 2019.

[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.

[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.

[42] Microsoft. Empowering the future of financial markets with london stock exchange group, 2022.

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

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

[45] Microsoft Azure. Confidential computing, 2024.

[46] NASDAQ. Nasdaq and aws partner to transform capital markets, 2021.

[47] Arista Networks. Arista 7132lb datasheet. https://www.arista.com/assets/data/pdf/Datasheets/7132LB-Datasheet.pdf.

[48] Arista Networks. Arista 7135lb datasheet. https://www.arista.com/assets/data/pdf/Datasheets/7135LB-Datasheet.pdf.

[49] John Osborne. High-frequency trading over leo, 2022.

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

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

[52] 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 *Proceedings of the Seventeenth USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, February 2020.

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

[54] Wikipedia. Algorithmic trading. https://en.wikipedia.org/wiki/Algorithmic_trading, 2024.

[55] 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 Fernando Kuipers and Ariel Orda, editors, *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*, pages 1–18. ACM, 2022.

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: Varying algorithmic trading technical analysis indicators expressed with Cuttlefish's MP handlers.

A Multi-threaded execution in virtual time

If a service provider wishes to increase overall virtual time throughput of the platform, in addition to using faster processors or hardware accelerators, Cuttlefish also supports scaling up the computation capacity for each MP through a multi-threaded variant. Each user provides an MP program with multiple handler functions to execute in parallel. The runtime workflow remains the same as described in §6, 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 can invoke the built-in service API, specifying a target thread ID and the data bytes for inter-thread communication.

Deterministic multi-threaded execution is naturally compatible with Cuttlefish's virtual time abstraction. In this variant, Cuttlefish operates in discrete epochs, each spanning VT virtual time cycles. Data scheduled for inter-thread communication during epoch k are batched and released at the start of the 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.