Hospitality.Watr.Exchange Whitepaper

Executive Summary

Hospitality.Watr.Exchange is a next-generation procurement and trading platform tailored for the hospitality industry. It enables restaurants and hospitality operators to source supplies more efficiently through three innovative exchange **rails**: (1) instant buy offers, (2) reverse-Dutch auctions, and (3) multi-party barter trades. By leveraging a modern event-driven architecture and advanced exchange mechanisms, Hospitality.Watr.Exchange promises to reduce procurement costs, increase supply chain agility, and minimize waste in inventory. The platform's high-level strategy focuses on delivering tangible business value: lowering the cost of goods for buyers, expanding market access for suppliers, and monetizing transactions through value-added services. Key performance indicators (KPIs) include procurement cost savings, faster fulfillment times, reduced stockouts, and improved supplier-buyer match rates.

At its core, the platform uses an **Aggregate-Transform-Persist-Publish (ATP)** architectural pattern to integrate disparate systems and data flows. All inbound purchase requests, bids, and trade offers are aggregated, transformed into a common **canonical event model**, persisted in a secure data store, and published to a pub/sub bus for processing by various domain microservices. This ensures a robust, scalable system where each component (sourcing, auctions, barter matching, compliance checks, accounts payable, etc.) can react to events in real-time and reliably reconstruct state from the event log if needed. The architecture is cloud-native (built on Microsoft Azure) and emphasizes security, compliance, and high availability.

For the CEO and COO of Toast (a leading restaurant technology platform), the business proposition is compelling: Hospitality.Watr.Exchange can unlock new revenue streams by facilitating more efficient purchasing for restaurant clients, while taking a transaction fee or subscription for premium analytics. By serving as an intelligent procurement exchange, Toast can deepen its relationship with its customer base – driving higher retention and creating a defensible ecosystem. For the CTO, the whitepaper provides a deep technical blueprint of how such a platform can be built and integrated, with emphasis on modular design, industry standards (like CloudEvents and EDI), and enterprise-grade reliability. The following sections dive into the business context, value metrics, and the full technical architecture of the solution.

Business Context and KPIs

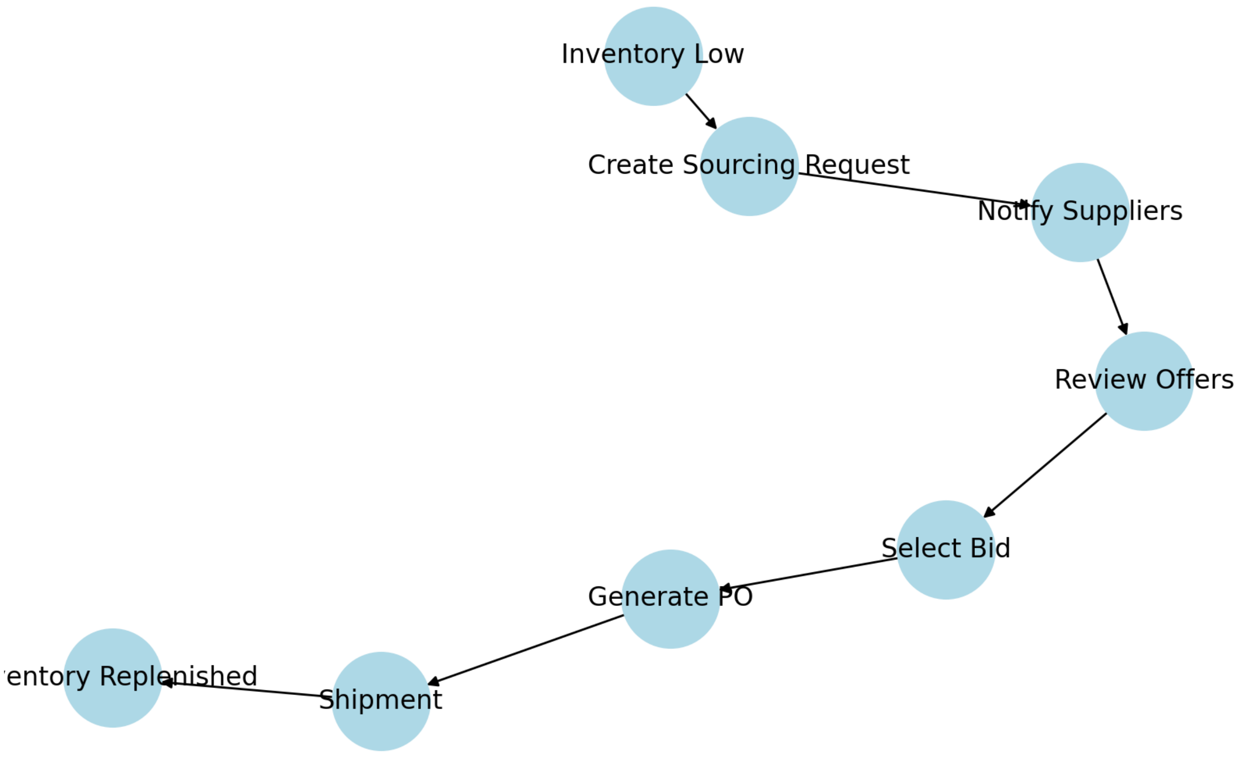
The hospitality industry (restaurants, hotels, foodservice) operates on thin margins, where supply costs and operational inefficiencies can make a big difference in profitability. **Toast**, as a major technology provider in this space, has an opportunity to extend its platform with a procurement exchange that directly tackles these pain points. Hospitality.Watr.Exchange is designed in this context – to streamline the way hospitality businesses source products (food ingredients, beverages, supplies) from distributors and manufacturers. By introducing modern exchange mechanisms, the platform aims to **improve price discovery, reduce waste, and ensure supply resilience**.

Value Proposition: Buyers (restaurants, hospitality operators) get access to a wider supplier network and dynamic pricing mechanisms to secure lower prices or favorable trade terms. They can buy instantly from standing offers, or initiate auctions to drive prices down, or even barter surplus goods in exchange for what they need. Suppliers gain a new channel to move inventory – whether selling immediately at posted prices, competing in auctions for big orders, or swapping products in barter deals to fulfill demand. The exchange guarantees neutral and efficient matching, so both sides benefit from greater market liquidity and transparency.

Monetization Strategy: The platform can monetize through transaction fees and premium services. For each completed trade (instant purchase, auction win, or barter cycle closure), Hospitality.Watr.Exchange takes a small commission. Additionally, a subscription model could be offered for advanced analytics, priority listing placement, or integration support (e.g. seamless ERP integrations and EDI handling for enterprise customers). Advertising or sponsored product placements in the marketplace is another potential revenue stream. Over time, data generated by the exchange (on pricing trends, demand forecasts) can inform new value-added offerings to suppliers and buyers, potentially creating **data-as-a-service** revenue.

Key Impact Metrics: The success of the platform will be measured by: - **Cost Savings for Buyers:** Reduction in procurement costs (e.g. restaurants saving on average 10-15% on key supplies by using auctions vs. traditional ordering). Reverse auctions historically have shown ~18-20% price reduction on average compared to static bids[1]. - **Supplier Market Reach and Throughput:** Number of new buyer relationships per supplier and increased sell-through of inventory (especially for excess or perishable stock via barter or discounts). - **Transaction Volume & Liquidity:** Total GMV (gross merchandise value) transacted through the exchange, number of trades per week, and average time to fill an order. Faster cycles (e.g. auctions completing in hours, instant buys in seconds) indicate efficiency. - **Operational Efficiency Gains:** Reduction in stockouts and waste. By leveraging reverse auctions and barter, restaurants can respond to **low inventory** situations quickly with dynamic sourcing (see Figure 1 below), avoiding emergency shortages. Suppliers can reduce waste by offloading excess products through barter or discounts. - **User Engagement & Retention:** Adoption metrics such as active buyer/supplier counts, repeat usage rates, and the attachment of exchange usage to Toast's core POS/management system (driving platform stickiness).

Ultimately, the exchange should demonstrate that it **lowers the total cost of ownership** for hospitality procurement and **improves supply continuity**, which translates to better margins and customer satisfaction in the hospitality business. These business outcomes will justify Toast's investment and align with its mission to empower hospitality operators.

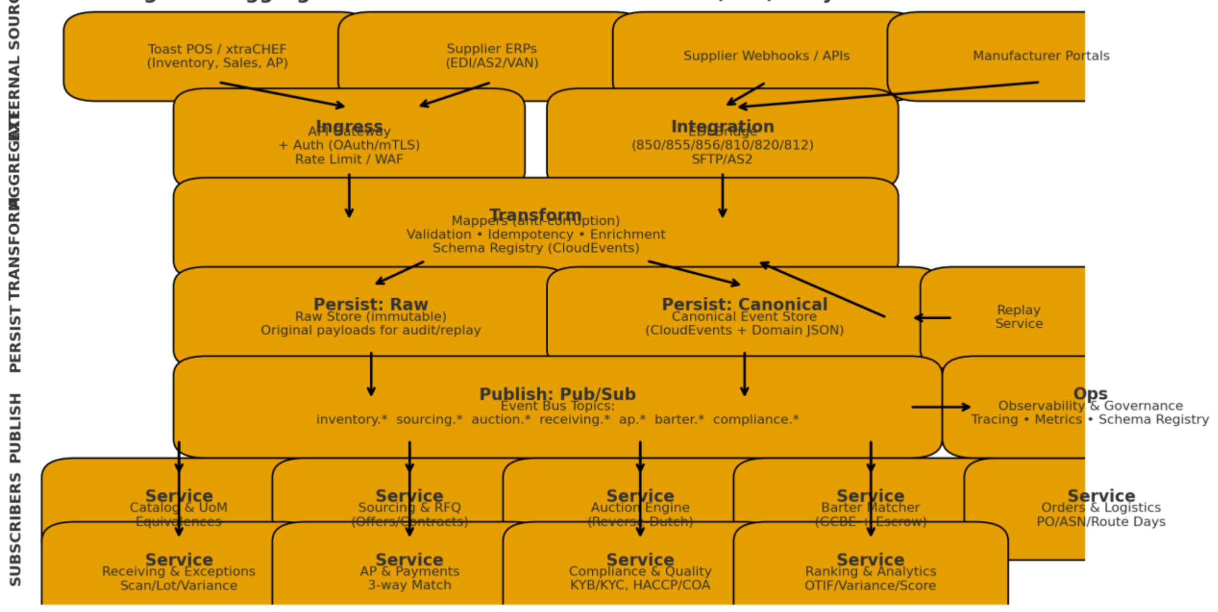


In this example process, a hospitality operator's **Inventory Management** detects low stock on an item, prompting the manager to create a sourcing request on the exchange. The **Supply Chain Marketplace** (Hospitality.Watr.Exchange) allows the manager to define requirements and list a request. Suppliers on the platform get notified and can propose bids or immediate offers. The manager reviews incoming offers and accepts the best bid. Upstream, a manufacturer may be involved (if the supplier needs to produce or source the item). The exchange coordinates the flow: once a bid is accepted, a purchase order is generated and sent to the supplier. The supplier fulfills the order (arranging shipment), and the restaurant's inventory is replenished. This collaborative process illustrates how the exchange connects hospitality buyers with suppliers/manufacturers seamlessly, replacing ad-hoc calls or emails with a structured, efficient workflow.

Architecture Overview (Aggregate-Transform-Persist-Publish)

The Hospitality.Watr.Exchange platform is built on a robust event-driven architecture following the **Aggregate-Transform-Persist+Publish (ATP)** pattern. This pattern ensures that data from various sources is ingested, normalized, stored, and disseminated reliably to all interested components. Figure 2 illustrates the high-level data flow of the ATP architecture within our platform.

Figure 2: Aggregate → Transform → Persist + Publish (ATP) — System Flow


EXTERNAL SOURCES
AGGREGATE
TRANSFORM
PERSIST
PUBLISH
SUBSCRIBERS
Toast POS / xtraCHEF (Inventory, Sales, AP)
Supplier ERPs (EDI/AS2/VAN)
Supplier Webhooks / APIs
Manufacturer Portals
Ingress
API Gateway + Auth (OAuth/mTLS) Rate Limit / WAF
Integration
EDI Bridge (850/855/856/810/820/812) SFTP/AS2
Transform
Mappers (anti-corruption) Validation • Idempotency • Enrichment Schema Registry (CloudEvents)
Persist: Raw
Raw Store (immutable) Original payloads for audit/replay
Persist: Canonical
Canonical Event Store (CloudEvents + Domain JSON)
Replay Service
Publish: Pub/Sub
Event Bus Topics: inventory.* sourcing.* auction.* receiving.* ap.* barter.* compliance.*
Ops
Observability & Governance Tracing • Metrics • Schema Registry
Service Catalog & UoM Equivalences
Service Sourcing & RFQ (Offers/Contracts)
Service Auction Engine (Reverse Dutch)
Service Barter Matcher (GCBE + Escrow)
Service Orders & Logistics PO/ASN/Route Days
Service Receiving & Exceptions Scan/Lot/Variance
Service AP & Payments 3-way Match
Service Compliance & Quality KYB/KYC, HACCP/COA
Service Ranking & Analytics OTIF/Variance/Score

Inbound data from multiple sources (e.g., a restaurant's procurement system, a supplier's application, or manual inputs) is **Aggregated** through a common interface. This could be an API Gateway or integration service that collects events such as a new purchase request or a bid submission from any channel. Next, events go through a **Transform** step where they are converted from source-specific formats into a unified **canonical schema**. For example, if a supplier's ERP sends an EDI 850 Purchase Order, it is transformed into a platform-specific PurchaseOrderCreated event in JSON. After normalization, events are **Persisted** in a **canonical store** – a database or event log that serves as the system of record. Finally, the events are **Published** on a pub/sub bus so that multiple subscribing services can react to them asynchronously. This decoupling means one event (e.g. an order created) can trigger inventory checks, supplier notifications, compliance validations, and analytics updates in parallel, without blocking the original transaction flow.

The ATP architecture provides several benefits: - **Loose Coupling:** Senders of data (e.g. an external ordering system) don't need to know which internal services will handle the event. They just drop an event into the system. Consumers subscribe to relevant events. This makes it easy to add new services or integrations without modifying the core flow. - **Data Consistency and Replay:** By persisting every event in a canonical log or database, we maintain an immutable history of transactions. Services can replay events from the store to recover state or backfill data, which is critical for resilience and debugging. If a downstream service was down, it can later read missed events from persistence to catch up. - **Scalability:** The aggregation layer can buffer bursts of incoming events (using queues) and the processing (transforms, etc.) can be scaled out via parallel worker processes. The publish/subscribe bus allows horizontal scaling of consumer services – multiple instances can consume different partitions of the event stream for throughput. - **Interoperability via Canonical Model:** By using a common data model (discussed in the next section) and a standard envelope like CloudEvents, the system ensures that all components interpret events uniformly. CloudEvents is an open specification providing a common structure (with standard fields like type, source, timestamp, etc.) for events across services[2]. Embracing CloudEvents means easier integration with cloud services and other platforms, since it's a widely supported format.

In summary, the architecture overview sets the stage: all interactions in the exchange are event-centric, flowing through the ATP pipeline. This guarantees that whether a user executes a **Buy Now**, an auction closes, or a complex barter trade is matched, the event is captured and propagated reliably to every part of the system that needs to know.

Sourcing Rails and Exchange Mechanisms

A unique aspect of Hospitality.Watr.Exchange is support for multiple **sourcing mechanisms (rails)** on the same platform. Each rail caters to different business scenarios but they are integrated under a common framework. The three rails are: (1) **Instant Buy / Standing Offers**, (2) **Reverse-Dutch Auctions**, and (3) **GCBE-style Barter**. This section describes each mechanism, how it works, and how it's implemented in the system.

1. Instant Buy / Standing Offers

The Instant Buy rail functions like a traditional marketplace or **posted-price exchange**. Suppliers can list **standing offers** for products – essentially catalog items with a price, available quantity, and fulfillment terms. Buyers (restaurant operators) can browse or search these listings and execute an immediate purchase (Instant Buy) if the terms are acceptable.

From a system perspective, when a buyer clicks "Buy Now" on a listing: - The client app (or Toast interface) sends a Purchase Order event into the system (e.g., `PurchaseOrderRequested` event with item, quantity, price). - This event goes through the ATP pipeline: aggregated via the API, transformed into a canonical order format, and persisted. - A **Sourcing Service** picks up the event from the bus and processes the order: it reserves the inventory from the specified supplier's standing offer, confirms pricing, and transitions the order to a booked state. - The supplier is notified instantly (via either the platform's portal or an EDI 855 Purchase Order Acknowledgment) confirming the order details. - Downstream events like `OrderConfirmed`, `InventoryAllocated` are published for other services to consume (e.g., the Receiving Service will expect an ASN, Accounts Payable will await an invoice event).

Instant Buy provides speed and certainty – it's most useful for routine purchases or when a buyer finds a good listed price. The platform ensures that standing offers remain updated (suppliers can programmatically update prices or quantities, even via APIs or an EDI 846 Inventory Advice message if needed). Monetization here is straightforward: the platform charges a small transaction fee on each Instant Buy order.

Monetary and Impact Consideration: Instant buys help when timing is critical (e.g., a chef discovers a staple ingredient is out of stock mid-week and needs to order immediately at market price). They may not always yield the lowest price, but they guarantee fulfillment. The availability of this rail ensures the exchange can serve as a reliable backup supply source in emergencies, thereby increasing the **supply resilience** KPI (fewer stockouts).

2. Reverse-Dutch Auctions (Descending Clock Auctions)

The second rail is a **Reverse Auction**, specifically a *reverse-Dutch auction*. In a typical **Dutch auction**, the price starts high and ticks downward until a buyer accepts the price. In a **reverse** scenario, roles swap: the buyer is the one soliciting bids, and multiple suppliers compete by offering lower prices until the auction closes. A *descending clock* format means the auction system continuously (or in stages) lowers the price, and suppliers can accept at a price point of their choosing; the first supplier to accept "wins" the auction at that price. If multi-lot procurement is needed, the auction can continue accepting multiple suppliers until the requested quantity is filled.

Process: When a buyer initiates a reverse-Dutch auction: - They specify the item, total quantity needed, and possibly a starting (ceiling) price and a floor price (lowest acceptable price), along with the auction duration. - The system broadcasts an `AuctionCreated` event with all these details. Suppliers subscribed to that product category receive the auction information (either through the platform's dashboard or an integration if they have automated bidding agents). - The auction countdown begins at the starting price. The platform can decrement the price in predefined intervals (e.g., drop by \$0.10 every minute) or continuously if using a real-time clock. - If a supplier is willing to fulfill the order at the current price, they send a bid acceptance. The first acceptance (for the full lot or a partial quantity in multi-lot case) is recorded. If the auction is multi-lot (e.g., buyer needs 100 units and suppliers each might supply a portion), the auction continues until the required quantity is met or time runs out, potentially accepting multiple winners in order of acceptance. - Auction events are published at each step: `PriceDropped`, `BidAccepted`, etc. When concluded, an `AuctionClosed` event captures the winning supplier(s) and final price(s).

Clearing Logic: In a single-winner descending auction, the clearing is straightforward – the first bid wins at that price. In a multi-unit scenario, it becomes a *descending multi-unit auction*: the price keeps descending and multiple suppliers can accept at various price points. The **pseudocode** for auction clearing would look like:

```
price = start_price
while current_time < end_time and quantity_remaining > 0:
    wait for any supplier to accept at current price or timeout for price tic
k
    if accept received:
        allocate quantity from that supplier
        quantity_remaining -= accepted_quantity
        record winner at current price
        if quantity_remaining <= 0: break
    drop price to next level
end_auction()
```

(See Appendix B for a more detailed pseudocode). If the auction ends with some quantity unfilled, the buyer can decide to accept partial fulfillment or trigger a new round. The platform might also enforce that once a supplier accepts, they commit to that quantity and price (no backing out), which is handled via immediate order creation events and perhaps escrow of the offer.

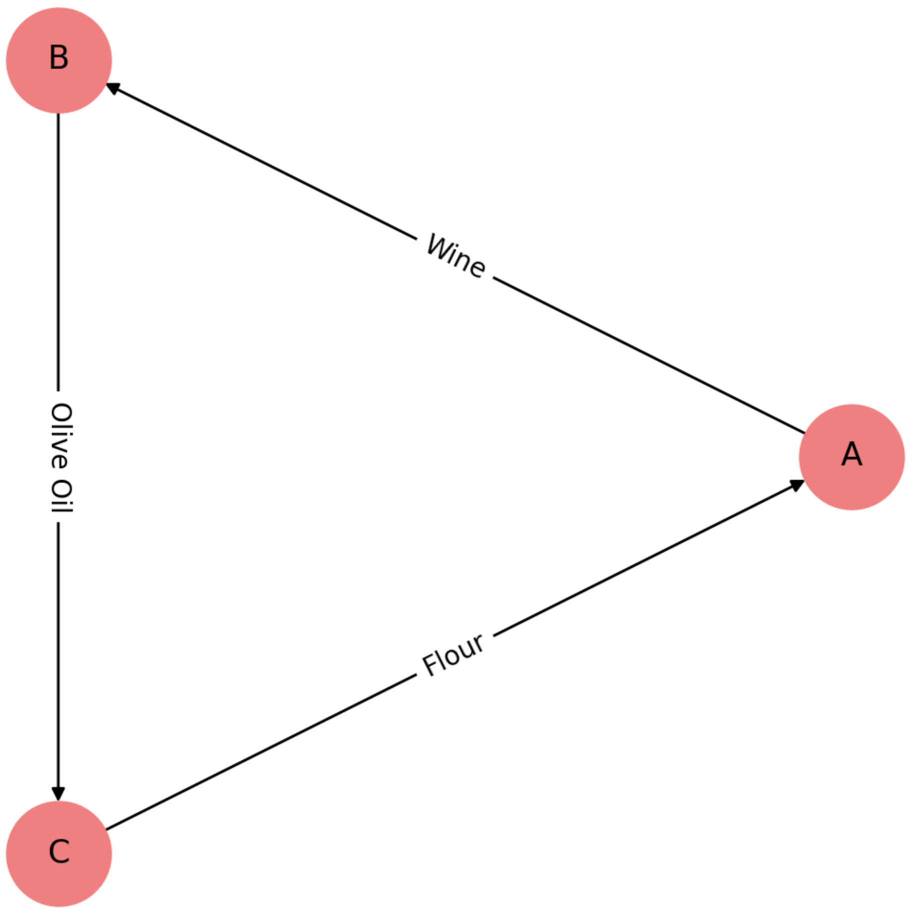
Business Benefits: Reverse auctions create **downward price pressure** by design, as multiple suppliers underbid each other[1]. This is especially powerful for commodity-like products or high-volume purchases, where an 18-20% cost reduction might be achievable on average for buyers[1]. It's also beneficial for suppliers who have cost advantages or excess inventory – they can win business by being the first to accept at a price slightly above their minimum. For example, if a restaurant chain needs a large quantity of chicken breast, suppliers in the region can compete; the one with a surplus or lower cost structure might grab the deal as soon as price drops within their acceptable range.

From a technical standpoint, implementing a descending clock auction requires **real-time event handling**. The clock ticks (price drops) can be modelled as events on a schedule, and the system must be able to handle nearly simultaneous accepts. The **Ranking Service** (discussed later) can also play a role here: if multiple acceptances come in very close together at a price point (especially in multi-winner scenario), some logic may rank which to honor first (likely by timestamp order). The system will publish the final results so that subsequent processes (like generating Purchase Orders, sending confirmations) happen automatically for the winning bids.

3. GCBE-Style Barter Exchange (Multi-Lateral Barter with Transformation)

The third rail is the **Generalized Commodity Barter Exchange (GCBE)** style barter trading. This is a novel mechanism allowing multi-party, multi-good barter transactions. Instead of using money as an intermediary, the system finds cycles of trades where each participant's needs are met by others' offers. For instance, Restaurant A has surplus wine and needs olive oil, Restaurant B has extra olive oil and needs flour, Restaurant C has surplus flour and wants wine – the platform could arrange a 3-way swap **A→B→C→A** that satisfies all parties without any cash changing hands.

GCBE is essentially a **graph-based algorithmic barter engine**[3]. Participants list offers of what they *have* (to give) and what they *want* (to receive). The engine then searches for a cycle or chain of exchanges: A gives what B wants, B gives what C wants, ... N gives what A wants, closing the loop[3]. All trades execute simultaneously to ensure no one is left holding unwanted goods mid-transaction. In computer science terms, it finds a cycle in a directed graph of wants/haves. The concept of "locks" is used – once a cycle is identified, all involved offers are locked and executed together, preventing any participant from backing out at the last second[4].



In this simplified diagram, three participants form a closed barter loop: Participant A transfers **Commodity X** to B, B sends **Commodity Y** to C, and C provides **Commodity Z** to A. Each gets the item they need by giving away something they have in surplus. The exchange platform's barter engine automatically identified this cycle and coordinated the swap. In practice, such cycles can involve many participants and commodities, but the principle is that **no money is exchanged**, only goods. The Hospitality.Watr.Exchange platform supports such cycles to allow creative trade of excess inventory or services (e.g., a hotel bartering unused food stock for cleaning supplies from another property).

To facilitate real-world barter at scale, the platform includes enhancements: - **Supplier Escrow:** When a barter cycle is identified, the platform acts as an escrow agent. Each party must commit (escrow) their goods – e.g., by verifying inventory availability or even physically moving goods to a neutral warehouse if needed – before the exchange executes. Only when all parties' commitments are secured does the system finalize the swap. This builds trust that no participant can default after receiving what they want. - **Transformation Services:** In some cases, a direct swap might not exist, but a *transformation* can bridge the gap. For example, Supplier X might offer a service to grind wheat into flour for a fee. If Restaurant A has wheat and needs flour, and Restaurant B has olive oil and needs wheat, a cycle could include Supplier X transforming A's wheat to flour that A ultimately receives, while A's wheat goes to X and perhaps olive oil flows to B, etc. The exchange can incorporate such supplier-provided transformations (with compensation terms) into barter deals. Essentially, this means some nodes in the barter graph can be *transformers* that convert one commodity to another. Those transformations are also locked in via smart contracts or escrow until completion.

Implementing GCBE matching is computationally more complex than bilateral trades. It involves solving a search problem (often akin to the classic "cycle detection in a directed graph" or even a combinatorial optimization if looking for the best rate or multi-cycle solution). The **matcher algorithm** (Appendix C provides a conceptual diagram or pseudocode) typically works like: 1. Collect all outstanding barter offers (nodes with their have/want). 2. Construct a directed graph where an edge from node A to B exists if A has something B wants (and the quantities/ratios match some acceptable rate). 3. Search for cycles in this graph (bounded by a certain length for practicality – e.g., find cycles up to 4 or 5 parties). 4. If found, evaluate if the cycle satisfies all quantity constraints and possibly optimizes a criterion (like maximizing the trade volume or best conversion ratios). Then lock and execute it – emit an `ExchangeCycleFormed` event detailing the multi-way trade. 5. Remove or update the involved offers and repeat (since one cycle execution may enable new possibilities or remove demands).

GCBE-style trades can dramatically improve efficiency by **eliminating the monetary middleman**, thus avoiding speculative or unnecessary markups[5][6]. Economically, such trades keep the exchange of value anchored in real goods, theoretically minimizing speculative "imaginary" value. In fact, by always collapsing trades into real exchanges, GCBE **keeps speculative intensity near zero** – participants get what they need for immediate use, rather than trading for money and potentially holding it[7][8]. This aligns with research indicating that multi-lateral barter can remove inefficiencies and even prevent certain market instabilities[9][10]. While the hospitality exchange's primary goal is practical (cost savings and waste reduction), it's notable that this design inherently promotes a stable market by focusing on real utility.

From a business standpoint, the barter rail is a differentiator. A restaurant might barter unsold perishable stock (e.g., excess dairy that might spoil) for cleaning supplies or paper goods with another restaurant that overstocked those – both sides save money and reduce waste. **Impact metrics** to track here include the value of goods exchanged via barter (which reflects cost avoided by participants) and waste reduction (if goods would have been discarded). The platform could even charge a smaller fee for barter transactions or a membership, since no money changes hands – perhaps a flat facilitation fee or subscription for participating in the barter network.

Vendor Loyalty Program with Blockchain Points

In addition to transactional efficiency, **Hospitality.Watr.Exchange** can strengthen long-term relationships between suppliers and tenants through a **blockchain-powered loyalty program**. The idea is to reward activity on the platform (purchasing, fulfilling orders on time, participating in auctions or barter cycles) with **loyalty points**. These points function like a **digital currency**: tradeable, transparent, and redeemable across the exchange ecosystem.

Program Design

- **Earning Points:**
    - Buyers earn points for consistent purchasing through the platform, bulk orders, early payments, and providing high supplier ratings.
    - Suppliers earn points for on-time delivery, auction participation, providing competitive bids, and offering barter transformation services.
    - Both earn points for platform growth actions such as onboarding new trading partners.
- **Point Tokenization:**
  Points are issued as **non-speculative, utility tokens** on a permissioned blockchain (e.g., Hyperledger Fabric, Quorum, or Azure Confidential Ledger). Each token represents one loyalty unit, with all issuance, transfers, and burns recorded immutably.
    - **Transparency:** Every participant can verify their point balance and history without trusting a central database alone.
    - **Interoperability:** Points can be traded peer-to-peer between vendors and tenants, enabling unique scenarios (a supplier might sell loyalty credits at a discount for immediate cash flow, or a tenant might buy points from another to reach a redemption threshold).
- **Trading Mechanism:**
  Loyalty points are listed as a special commodity in the exchange. Vendors or

tenants can post offers (sell points for cash or barter points for goods). The **barter rail** can incorporate points as a leg in a cycle: e.g., Restaurant A barters surplus wine to Supplier B, Supplier B sends olive oil, and Restaurant A pays in part with loyalty points.

Redemption & Incentives

- **Redeem for Discounts:** Points can offset future order costs (e.g., 100 points = $10 discount on next purchase).
- **Exclusive Access:** Vendors may offer "points-only" promotions (e.g., early access to scarce items, free delivery for X points).
- **Services & Analytics:** Toast can allow points to be redeemed for premium analytics subscriptions, API access, or compliance services.
- **Financial Integration:** In future phases, points could be integrated with Toast's payment infrastructure—allowing mixed settlements (cash + points).

Blockchain Benefits

- **Trust:** Immutable ledger ensures fairness—no party can inflate their points or tamper with history.
- **Auditability:** Regulators or auditors can verify loyalty issuance and redemption, which is critical if points are considered a financial liability on supplier balance sheets.
- **Portability:** If approved by Toast, points could extend beyond Hospitality.Watr.Exchange to other Toast ecosystem services, creating a unified loyalty economy.
- **Programmability:** Smart contracts can enforce rules automatically (e.g., points expire after 12 months, double points promotion during off-peak months).

Business Impact

- **Increased Engagement:** By tying tangible, tradeable rewards to activity, vendors and tenants have stronger incentives to transact on-platform instead of off-platform.
- **Sticky Ecosystem:** The blockchain-based loyalty system creates a "walled garden" where participants accumulate value that is most useful when reinvested in the Toast ecosystem, driving retention.
- **New Monetization:** Toast can monetize loyalty points by selling them in bulk to suppliers (who then distribute to tenants) or by taking a small fee on point trades in the marketplace.

- **Network Effects:** As points gain liquidity, they become a medium of exchange in their own right within hospitality procurement—further cementing Hospitality.Watr.Exchange as the central hub.

Canonical Event Model (CloudEvents and Domain Schema)

All activities in the exchange are represented as **events** in a standardized format. We adopt the **CloudEvents 1.0 specification** as the envelope for all our events to ensure consistency and interoperability. CloudEvents provides common metadata attributes – such as `id`, `source`, `type`, `time` – to describe an event, along with a `data` payload for domain-specific details[11][12]. By using CloudEvents, internal services and external integrators have a uniform way to parse and interpret messages, and we leverage existing tooling in Azure (Azure Event Grid, Service Bus) that supports this schema.

Within the CloudEvents envelope, we design a **domain schema** for each major event type in our system. Key domain events include: - **PurchaseOrderRequested** – when a buyer submits an order (either via instant buy or after an auction). Data might include order ID, buyer info, supplier (if known), item list, quantities, and prices. - **BidPlaced** – when a supplier places a bid in an auction, or **OfferAccepted** – when a supplier accepts the current price in a reverse-Dutch auction. Data includes auction ID, supplier ID, accepted price, quantity. - **AuctionCreated**, **PriceDropped**, **AuctionClosed** – events to manage auction state. For example, AuctionClosed would detail the winners. - **BarterOfferCreated** – when a participant posts what they have and want for barter. Contains offer ID, owner ID, offered commodity and quantity, requested commodity (or multiple via legs) and ratio, expiration, etc. - **ExchangeCycleFormed** – when the GCBE engine finds a valid cycle and it executes. Data would enumerate the participants and the goods flow for each (like A gave X quantity of item1 to B, B gave Y of item2 to C, ...). - **ShipmentDispatched (ASN)** – representing an Advance Ship Notice (EDI 856 equivalent) when a supplier ships goods. Data: order ID, carrier info, tracking, contents, expected arrival. - **OrderDelivered / Received** – confirmation that the buyer received the items (could be triggered by a receiving scan, etc.). - **InvoiceIssued** – an invoice event (EDI 810 equivalent) from supplier to buyer for an order or a settlement of a barter value difference (if any). - **PaymentSent** – a payment event (EDI 820) from buyer to supplier, or **CreditAdjustment** (EDI 812) if there's any post-invoice adjustment.

All these events share some common structure (thanks to CloudEvents) but have custom `data` sections. We maintain a **schema registry** or documentation (Appendix A provides examples of canonical schemas for key events). For instance, a `PurchaseOrderRequested` event's data might look like:

```
{
  "orderId": "12345",
  "buyer": { "orgId": "REST123", "name": "Great Restaurant" },
  "supplier": { "orgId": "SUP456", "name": "FoodSupplierCo" },
  "items": [
    { "sku": "1001", "description": "Olive Oil - 1L", "quantity": 20, "unitPr
ice": 5.00 }
  ],
  "totalPrice": 100.00,
  "exchangeMechanism": "InstantBuy",
  "originalEvent": "AuctionClosed"
}
```

This JSON would be inside the CloudEvent `data` field, with `type` attribute perhaps set to `com.hospitalitywatr.order.requested`. Similarly, an Auction event would have type `com.hospitalitywatr.auction.closed` and carry details of winners.

Using a canonical model has immediate advantages for **integration**: Toast's existing systems or third-parties can subscribe to certain event types and be assured of the format. If a large buyer wants to ingest every `InvoiceIssued` event into their accounting software, they know the schema in advance. It also simplifies **testing** and **validation** – we can define JSON Schemas for each event type and ensure producers/consumers conform.

CloudEvents & Azure: The choice of CloudEvents aligns with Azure's Event Grid and Service Bus capabilities. Azure Event Grid has native support for CloudEvents, meaning our events can flow through Azure's infrastructure without needing custom envelope mapping[2][13]. We also gain the possibility of easily extending the system or integrating serverless components (Azure Functions) by simply subscribing them to particular event types.

Additionally, the canonical event model is designed to carry EDI references where needed. For example, a `ShipmentDispatched` event might include an EDI control number or reference that maps to an 856 message, enabling easier correlation with external documents. This brings us to the next topic: integration with traditional EDI workflows.

Domain Services and Microservices

The platform is composed of several **domain-aligned microservices**, each responsible for a specific business capability. These services communicate primarily via the event bus (by publishing and subscribing to events) and occasionally through direct APIs when synchronous operations are required (though we favor eventual consistency via events).

Key domain services include:

- **Sourcing Service:** Handles creation of orders and matching with offers. When a `PurchaseOrderRequested` event comes in, this service verifies the request (does the supplier have a standing offer? did the auction produce a valid winner?), reserves inventory, and orchestrates order confirmation. For instant buys, it directly creates the order with the chosen supplier's offer. For auctions, it waits for `AuctionClosed` events to then generate the actual orders to winners. This service also can expose APIs for searching offers or listing products.

- **Auction Service:** Manages the lifecycle of reverse auctions. It handles events like `AuctionCreated` (initializing internal state/timers), runs the price decrement logic (publishing `PriceDropped` events on schedule), and processes incoming bids or acceptances. On receiving a `BidPlaced` event (or an acceptance trigger), it decides if the auction is won or continues. At auction end, it emits `AuctionClosed`. This service ensures the auction rules are enforced (time boxing, price floor, no bids after close, etc.). It may also produce real-time status updates to clients via WebSocket or SignalR notifications so that bidders see the current price in real-time.

- **Barter/Exchange Matching Service:** Runs the **GCBE algorithm**. It listens for new barter offers (`BarterOfferCreated` events) and maintains an in-memory (or cached) graph of all outstanding offers. Periodically or upon significant changes, it attempts to find match cycles. When a cycle is found, it coordinates with the involved parties: possibly sending each a confirmation request or just automatically executing if pre-authorized. It then publishes an `ExchangeCycleFormed` event. This service handles multi-leg logic and also ensures partial fulfilment rules (if someone offered to barter 100 units but the cycle uses 50, it may update their remaining offer). Because barter matching can be intensive, this might be implemented to run in batch windows or use specialized algorithms to be efficient.

- **Receiving & Logistics Service:** Focuses on the fulfilment side. When shipments are dispatched, this service handles `ShipmentDispatched` (ASN) events. It can update the status of orders in the system and notify buyers of incoming deliveries. It might integrate with carrier tracking APIs. It also listens for `OrderDelivered` or receiving confirmations – possibly triggered by scanning a delivery, which might come as an event or API call from the Toast tablet when goods arrive at a restaurant. This service essentially closes the loop on orders, ensuring that deliveries are logged and feeding that info to other parts like AP (accounts payable).

- **Compliance and Verification Service:** The exchange likely needs to ensure that trades meet certain compliance rules. For example, if alcohol or controlled substances are traded (like a restaurant bartering wine), regulatory checks are needed (proper licenses, age verification of receivers, etc.). This service subscribes to events like `PurchaseOrderRequested` or `ExchangeCycleFormed` and runs rule checks. It could cross-check participants against compliance requirements, block or flag transactions that violate policies (publishing a `TradeComplianceFailed` event if so). It also ensures any financial transactions (like payments) meet security standards (e.g., PCI compliance if handling payments, though likely payment is outside the scope of the exchange directly – more on that in security).

- **Ranking and Analytics Service:** Although named "Ranking", this service has a dual role. First, for any scenario where the platform needs to **rank offers or suppliers**, it provides that logic. For example, in auctions, if multiple bids arrive, it may rank them not just by price but also by supplier rating, quality, or lead time. This correlates to a **scoring auction** concept where offers are evaluated on multiple attributes[14][15]. The ranking service could compute a composite score (with weights for price, vendor rating, etc.) and help the Sourcing or Auction services pick the best overall offer. Second, this service performs analytics: aggregating data from events to generate insights – average purchase prices, supplier performance metrics (on-time delivery, win rate in auctions), demand trends for certain commodities, etc. These analytics can feed back into business decisions (like adjusting how auctions are structured or identifying opportunities for new suppliers to join for popular items).

- **Accounts Payable (Settlement) Service:** Handles the financial side of trades. When an `InvoiceIssued` event is published (say from a supplier's system or generated by the exchange for the supplier), this service matches it to the original order and prepares the payment. It might integrate with payment systems or ERP. If integrated with Toast's payment processing, it can trigger or schedule an ACH or credit card payment to the supplier (or at least mark that payment is due). On receiving a `PaymentSent` (820) event, it closes the loop by updating records that the order is paid. For barter transactions, this service might also handle any cash equalization if needed (e.g., if one party gave more value than they received, a monetary credit might be arranged – the service would issue an 812 adjustment event or invoice for the difference).

These domain services are **loosely coupled** via the event bus. For example, the Auction service doesn't directly call the AP service; it simply emits who won and at what price. The AP service, upon seeing an AuctionClosed event, knows an invoice will be needed eventually. The independence of services also means each can be developed, scaled, and deployed independently, which is aligned with microservice best practices.

Integration and EDI Message Flow

Many hospitality companies and suppliers still rely on **EDI (Electronic Data Interchange)** messages to conduct business. To seamlessly integrate with existing ERP and procurement systems, Hospitality.Watr.Exchange supports the standard EDI transaction set for purchase-to-pay processes. The platform essentially **bridges the real-time exchange world with the EDI world** by translating events to EDI messages and vice versa where needed.

The typical EDI message flow, mapped to our event model, is as follows:

1. **Purchase Order (EDI 850)** – When a buyer's system issues a PO, it can be sent to the exchange as an EDI 850. The integration layer of our platform will translate the EDI 850 into a `PurchaseOrderRequested` event (populating all the data from the EDI fields). Conversely, if an order is initiated on the platform (say via the UI or API), the platform can generate an EDI 850 to send to a supplier that prefers traditional EDI. Essentially, 850 is the *creation of an order*. The exchange ensures that each 850 becomes a corresponding event in the canonical model.

2. **Order Acknowledgment (EDI 855)** – After a supplier receives a PO, they typically respond with a 855 PO Acknowledgment (confirming acceptance, or detailing any changes/backorder). Our system can consume a 855 (transforming it into an `OrderAcknowledged` event that updates order status). If the supplier is fully on the exchange platform, this might be done through a direct event (e.g., supplier clicks "Accept Order" which triggers the same event). But to integrate with those on EDI, the exchange's EDI gateway sends/receives these messages accordingly. The 855 ensures the buyer and the exchange know the order is confirmed by the supplier.

3. **Advance Ship Notice (EDI 856)** – Before shipping or at dispatch, the supplier sends an ASN (856) indicating what is being shipped, when, and how. This is critical for the buyer to prepare receiving. The platform will convert 856 into a `ShipmentDispatched` event. This event then triggers our Receiving Service to expect a delivery. We might also convert events into an outbound 856 if needed (for instance, in a barter where multiple parties ship, each might get an ASN from the platform for what to send to whom).

4. **Invoice (EDI 810)** – After goods are shipped, the supplier issues an invoice (810). The exchange can ingest an 810 message and turn it into an `InvoiceIssued` event linked to the original order. If the supplier uses the platform's UI instead to create an invoice, we can likewise generate an EDI 810 out to the buyer's accounts payable system. Essentially, the 810 marks the financial obligation for the buyer.

5. **Payment Order/Remittance (EDI 820)** – The buyer's payment to the supplier can be represented as an 820 (remittance advice or payment order). If the platform handles payments, it might generate an 820 to tell the supplier "we have initiated payment of X amount for invoice Y". Or the buyer's ERP might send an 820 to the exchange, which we translate into a `PaymentSent` event. This closes the loop financially.

6. **Credit/Debit Adjustment (EDI 812)** – If there are any issues (returns, price adjustments, etc.), an 812 is used. The platform would handle this by events like `TradeAdjustment` or `CreditIssued`, and can produce or consume 812 messages accordingly. For example, if a restaurant received spoiled goods and negotiates a credit, an 812 can be issued by supplier and the exchange logs that as an adjustment event.

All these flows ensure that a company that isn't fully onboarded to the exchange UI can still participate using their backend systems. **Toast's integration** with the exchange could mean that for restaurants using Toast, their purchasing module or inventory system directly emits events (no EDI needed on their side), but if their supplier is not on the platform, the exchange converts those events to EDI for the supplier. Vice versa, a supplier on SAP Ariba could send an EDI and the exchange will consume it.

The **EDI Integration Service** within our platform is responsible for this translation. It likely uses mapping templates (e.g., mapping 850 fields to our JSON event schema fields). It also manages EDI envelopes, acknowledgments (like 997 functional acknowledgments) to ensure trading partners trust the pipeline. This service might use a library or product (like Azure Logic Apps with an EDI connector, or a third-party integration tool) to reduce custom coding.

From a **messaging architecture** viewpoint, each EDI message can be thought of as an alternative representation of an event: - Inbound EDI is parsed and then a CloudEvent is published internally. - Outbound events destined for EDI partners are captured and an EDI file is constructed and transmitted via VAN or AS2 accordingly.

By supporting these EDI transactions (850/855/856/810/820/812), Hospitality.Watr.Exchange can fit into existing supply chain processes without forcing every participant to immediately change their IT systems. It eases adoption, which is key to building network liquidity quickly.

Data Model Overview

The platform's data model underpins how information is stored in the canonical store and used across services. While much of the state is event-sourced, we also maintain queryable projections for fast lookups (for example, a list of active offers or open orders). Here we provide an overview of key entities and relationships in the data model:

- **Organization (Org):** Represents a company or business entity (could be a restaurant, a supplier, a manufacturer, etc.). Attributes include org ID, name, type (buyer/supplier/both), location, and any registration details. Orgs have users (for authentication/permissions) but those details are in an Identity management system, not the core exchange data model.
- **Product/Commodity:** A master catalog of items that can be traded. Each has an ID (maybe UPC or a platform-specific code), description, unit of measure, category, etc. Commodities may also track attributes like shelf-life or whether they require special handling (for compliance checks). For the barter system, commodities form the basis of what's offered/wanted.
- **Offer:** This entity captures any *standing offer* from a supplier (for instant buy). Fields: supplier org, product, price, available quantity, expiration or review date. Offers can be marked as published (visible to buyers) or private if needed. These are essentially what populate the marketplace listing.
- **Auction:** Captures an ongoing or past auction. Fields: auction ID, buyer org (initiator), product and total quantity, start price, current price (if ongoing), status (open/closed), start and end times. There might be a related table of AuctionBids or acceptances, or these might be kept transiently until close. For multi-lot auctions, the Auction record remains but winners are linked via orders.

- **Barter Offer (Ask):** In line with GCBE, an Ask/Offer in barter context might be broken into parent and leg structures as seen in the reference design[16][17]. A parent Ask could have multiple legs (sub-offers) if someone offers multiple items or will accept multiple alternative items. Fields for a barter ask: offer ID, owner org, list of commodities they *have* (with quantities) and *want* (with desired quantities or ratio). In the generalized model, ratio fields indicate the rate of exchange between what they have and want (like 1 barrel of oil for 3 cases of wine, etc.)[18][19]. We also keep availability and whether partial fills are allowed[20][21]. Each ask may have a validity (time window) and a max cycle length it will participate in (some might not want extremely long barter chains). A *graph* of these asks is constructed for matching.

- **Order:** Represents a committed transaction of goods from a seller to a buyer. Fields: order ID, buyer org, seller org, order date, status (pending, shipped, delivered, etc.). It will have line items referencing products and quantities, and pricing (for monetary trades). For barter trades, an order might still be generated for each bilateral leg of the cycle (for record-keeping, e.g., each pair of orgs exchanging goods might get an order record with $0 price but listing goods to send/receive, or at least a delivery note). Orders also reference related events or source (like which auction or barter cycle led to them, via foreign keys or tags).

- **Shipment:** A record of a shipment (ASN equivalent). Fields: shipment ID, order ID (could link to one or multiple orders if consolidated), carrier, shipment date, expected delivery date, tracking number, and a manifest of items (which should match order line items quantities). Shipments have statuses as well (in-transit, delivered).

- **Invoice:** A billing record linking to an order (or multiple orders if combined billing). Fields: invoice ID, supplier, buyer, amount, invoice date, due date, paid status. In a barter, invoices may be zero (if perfectly even exchange) or might be used to settle differences.

- **Payment:** A record of payment transactions. Fields: payment ID, from org, to org, amount, date, method (ACH, credit card, etc.), and related invoice ID. If Toast handles payment processing, these records might sync with Toast's payment subsystem.

- **User / Identity:** Not a focus of the whitepaper, but each action is done by an authenticated user belonging to an org. We likely integrate with Toast's user accounts. The data model would reference user IDs on relevant records like who posted an offer or confirmed a delivery.

- **Event Store:** While not an entity per se, it's worth noting the canonical event store holds events which can be considered the source of truth. We will have event types as described, each event referencing the relevant entities by ID. Some services build **projections** (tables or materialized views) out of these for quick reads. For instance, a “Live Order View” that composes info from Orders, Shipments, Invoices to display to a user current status.

In an **Azure context**, the data model could be implemented via a mixture of Azure SQL for structured data (like offers, orders which benefit from relational consistency), Cosmos DB for flexible event storage or queryable JSON of events, and Blob Storage for large payloads or archival (like storing original EDI files or large attachments). We might also use Azure Data Lake or Synapse for analytics on events history.

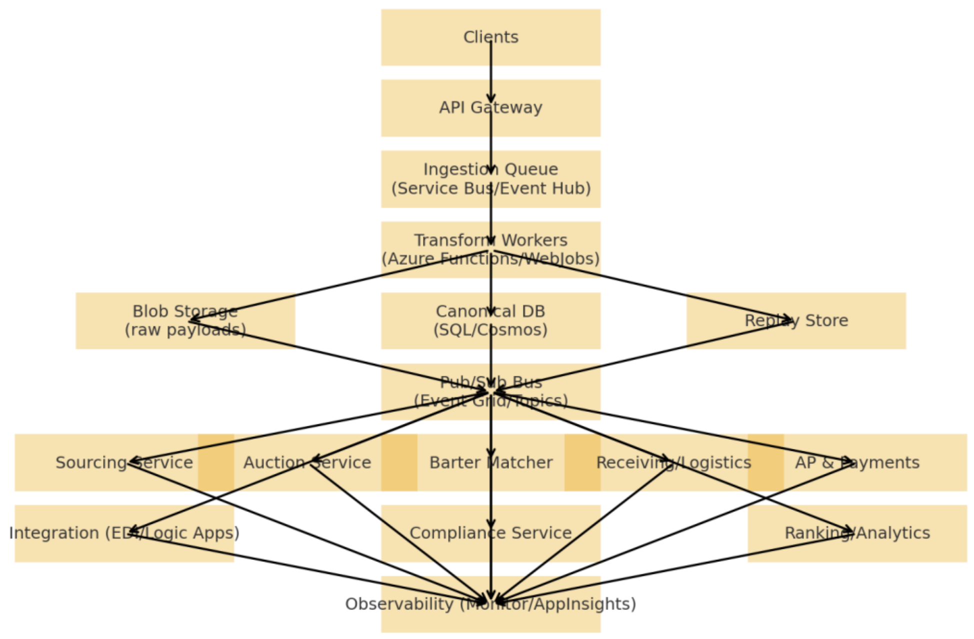
Appendix A will present sample schema definitions or table layouts for some of these, such as a simplified DDL for Orders and Asks (barter offers). For example, a snippet from the prototype barter exchange data model shows an **Ask** table and **AskLegs** table[16][17] – which aligns with allowing multi-commodity offers, with ratios for each leg. Our implementation may simplify or adjust that model, but the principle of splitting multi-item offers into legs is retained for flexibility.

Overall, the data model is designed to accommodate both **transactional workflows and search/analytics**. We will ensure key fields are indexed (like product IDs, org IDs) so the system can quickly find, say, all open offers for “olive oil” or all auctions initiated by a certain restaurant. Using a canonical data model and event sourcing also means any service can reconstruct the state or join the event stream at any point to build new projections, making the system extensible and robust against data loss in any single service.

Deployment Architecture (Azure-Native Implementation)

The platform will be deployed on Microsoft Azure, leveraging native services for scalability, security, and manageability.

Figure 4: Azure Deployment Architecture (Reference)


Clients
API Gateway
Ingestion Queue (Service Bus/Event Hub)
Transform Workers (Azure Functions/WebJobs)
Blob Storage (raw payloads)
Canonical DB (SQL/Cosmos)
Replay Store
Pub/Sub Bus (Event Grid/Topics)
Sourcing Service
Auction Service
Barter Matcher
Receiving/Logistics
AP & Payments
Integration (EDI/Logic Apps)
Compliance Service
Ranking/Analytics
Observability (Monitor/AppInsights)

4 illustrates a reference deployment architecture.

The diagram shows key components in a cloud deployment and how they interact: - **API Gateway:** All external calls (from Toast's UI or partner systems) pass through an API Gateway (e.g., Azure API Management or Azure Front Door with routing to microservices). This gateway provides a unified endpoint, handles authentication (e.g., OAuth tokens for org/users), throttling, and routing to the appropriate service endpoints or queues. - **Ingestion Queue:** For event-driven ingestion, an Azure Service Bus or Azure Event Hub is used as a **message queue**. After the API Gateway receives a request (say an HTTP POST for a new Order or a supplier's bid), it can forward the payload into a Service Bus **queue or topic** for processing. The queue decouples the intake from processing, enabling reliable buffering. - **Transform Workers (Azure Functions or WebJobs):** These are the **Aggregate/Transform** components. They pull messages from the queue. For example, one Azure Function triggers when a new message arrives in the "orders-incoming" queue, transforms the raw JSON or EDI into a canonical event, and then writes it to the persistent store and publishes to the bus. We might have separate function sets for different data (one for EDI-to-event transformation, one for processing UI-driven events, etc.), but conceptually they ensure all inbound data becomes a CloudEvent. - **Blob Storage and Database (Canonical Store):** We use Azure Blob Storage for storing raw events or documents (like storing the original EDI files or large JSON event payloads) and an Azure SQL/Cosmos DB for structured representation. The **Persist** step in ATP would likely write to an **append-only event table** or Azure Cosmos container, keyed by event ID. This acts as our event log. In addition, structured tables for things like Orders, Offers, etc., can be updated via an **Event Sourcing** consumer or directly by the service if using a more traditional approach for some items. Blob Storage is also used for any file exchange (if a user uploads a spreadsheet of items to barter, it could be stored and referenced). - **Pub/Sub Event Bus:** Azure Service Bus Topics or Azure Event Grid serve as the **Publish/Subscribe** backbone. Once an event is persisted, it is published on a topic that all interested services subscribe to. For instance, an Event Grid topic "ExchangeEvents" could broadcast events to multiple subscribers (the microservices). Azure Event Grid is well-suited here, as it natively uses a pub/sub model and supports direct subscription by Functions, Logic Apps, or any webhook endpoints. Each microservice could have a subscription filtered by event type (e.g., Compliance Service subscribes to all `PurchaseOrderRequested` events, AP Service subscribes to `InvoiceIssued` and `PaymentSent`, etc.). This ensures events reach the services that need them reliably. - **Microservices (Domain Services):** Each domain service (Sourcing, Auction, Barter, Receiving, Compliance, Ranking, AP) can be implemented as a set of Azure **App Services** or containerized microservices (in Azure Kubernetes Service, AKS). They listen to the event bus (via Event Grid subscriptions or Service Bus subscriptions). They also expose REST/gRPC endpoints for any synchronous operations or queries (for example, a UI querying current auctions might call the Auction Service's API to get the latest bids snapshot). These services often will update a **projection database** (like a Cosmos DB container or SQL table) that holds their view of data (for quick reads). For instance, Auction Service might maintain a table of current auctions and their latest price, updated every time a bid event comes in. - **Integration Services:** For EDI and external integration, an Integration Service (could be Azure Logic Apps or Functions with B2B connectors) handles communication with external systems. It might drop EDI files to an SFTP or use AS2 to send/receive messages with trading partners, then push those into our system as events. On the flip side, it takes internal events and formats them to EDI outbounds. - **Monitoring and Logging:** Azure Monitor, Application Insights, and Azure Log Analytics are used across the board to track performance, errors, and business metrics. Each microservice emits logs and metrics that can be aggregated (e.g., auction duration, number of bids, etc., are tracked). - **API for Toast Platform:** The exchange provides APIs/SDK so that Toast's existing products can interface. For example, a Toast Inventory module could call an API on Hospitality.Watr.Exchange to query available offers or start an auction when stock is low (as in our earlier BPMN scenario). These APIs go through the same gateway and adhere to security.

This Azure-native design ensures we get **high availability** (through Azure's regional replication and failover capabilities) and can leverage PaaS services to reduce operational overhead. For instance, using Azure Functions for event handling means auto-scale is built in (Functions can spin out multiple instances on high event volume). Azure Service Bus guarantees **delivery at least once** for messages, and we design idempotent handlers so that even if an event is delivered twice, our state updates only once.

Additionally, **containerization** is an option for core services if needed (AKS), but if we can design most as stateless Functions or App Services with managed backing stores, the deployment is simplified.

Security aspects (next section) are also addressed through Azure features: the API gateway can enforce OAuth with Toast's identity provider, and services can use Managed Identities to securely access databases and the event grid.

In summary, the deployment architecture combines the reliability of event-driven design (queuing, pub/sub) with the elasticity of cloud services, ensuring that Hospitality.Watr.Exchange can operate 24/7, scale during peak procurement seasons (for example, holidays when order volumes spike), and integrate easily with both modern and legacy systems in the hospitality supply chain.

Security and Compliance Considerations

Operating a procurement exchange involves sensitive data and critical business processes, so security and compliance are paramount. We address security at multiple levels:

1. Authentication & Authorization: Only authorized users and systems can access the platform's APIs and data. Toast's single sign-on and user management can be leveraged so that restaurant operators use their Toast accounts to access the exchange. The API Gateway requires OAuth 2.0 tokens (e.g., using Azure AD B2C or another identity platform integrated with Toast credentials). Each API call maps to a specific role permission – for example, only a user with a buyer role in an organization can create an order event for that organization. Similarly, suppliers have access only to their offers, bids, and so on. Within the event bus, services authenticate with Azure Managed Identity or SAS keys to ensure only our services subscribe/publish, preventing any rogue listener.

2. Data Encryption: All data in transit is encrypted via HTTPS or AMQP over TLS for the messaging. Data at rest in Azure is encrypted by default (Storage Service Encryption for blobs, Transparent Data Encryption for SQL, etc.). Additionally, sensitive fields (like payment details, if any, or personal user info) can be encrypted at the application level. If any credit card processing is involved, the platform won't store card numbers directly but rather integrate with Toast's PCI-compliant systems, ensuring we remain PCI DSS compliant.

3. Network Security: We use Azure Virtual Networks and private endpoints for internal communication between services and data stores. The API gateway is the only public entry point; everything else (Functions, databases) can be isolated. Integration endpoints for EDI (like an SFTP server or AS2 endpoint) are secured and likely whitelisted to known trading partner IPs. Azure provides DDoS protection by default, and we can enable a Web Application Firewall (WAF) on the gateway to filter malicious traffic.

4. Compliance Standards: The hospitality industry might not have specific data privacy regulations beyond standard ones (like GDPR if dealing with any personal data, or possibly some FDA regulations if dealing with food safety data). We ensure GDPR compliance by protecting any personal data (like user contact info) and providing means to delete/anonymize it if required. For financial data, **SOX compliance** might be relevant to Toast as a public company – our logs and records (like who approved what trade, and audit trails for orders) will be maintained in tamper-evident ways (the event store provides an immutable history, which is great for auditing). If the exchange handles any payments or bank info, we also adhere to PCI DSS and possibly NACHA rules for ACH. However, likely we offload actual payment to existing Toast payment infrastructure which already complies, keeping our scope smaller.

5. Smart Contract and Escrow Security: In the barter process, when acting as an escrow, the platform may hold commitments or even physical custody via third parties. We'd implement **smart contracts** or at least contractual rules encoded in the system – e.g., if all parties confirm the goods are ready, then trigger the swap; if any party fails to deliver, unwind the trade or compensate others. These rules must be transparent to users. Security here means ensuring one party cannot cheat – the system design (locking and all-or-nothing execution) protects against that. If using blockchain or distributed ledger for added trust (as an optional future step), that could further secure and audit barter transactions, but even without that, the platform's centralized escrow logic is designed to be neutral and enforced by code.

6. Confidentiality and Data Segmentation: Each organization's data should be isolated such that one supplier can't see another's private info (offers or bids) unless intentionally part of the same transaction. Access control on events and queries is enforced. For instance, a bid event might be public to the auction participants but detailed info like a supplier's identity could be masked to other bidders until the auction ends to prevent collusion. We can use techniques like assigning participant IDs in auctions instead of real names, known only to the system and revealed after if needed.

7. Compliance Logging: All key actions (like who accepted a bid when, who posted an offer) are logged. The system can produce compliance reports – for example, traceability for food products (which supplier provided a batch in a barter chain, etc., useful for recalls or safety compliance). The exchange could integrate with food safety databases or certification checks (ensuring a supplier of alcohol has a valid liquor license on file – compliance service can verify such things).

8. Regulatory considerations: If the exchange at some point deals with financial instruments (less likely here) or cross-border trade, we might consider legal requirements (like customs docs, or being a licensed money transmitter if handling payments for barter credits). At launch, we keep it simpler: the exchange facilitates trade but payments are between buyer and supplier (just mediated), so we might avoid money transmitter regulation. For barter, if any **tax implications** (barter is often taxable as income), we can provide reports to participants to aid in compliance.

In conclusion, security and compliance are woven into the architecture: - Azure AD for identity, encryption everywhere, strong network isolation. - Event-driven design also helps security: we can centrally audit all events, and apply policies. For example, a compliance rule could be “if a purchase of more than \$10k is made, require managerial approval” – this can be a service that listens and possibly pauses the order by not forwarding it until approved (publishing an event for approval needed). All of these measures ensure that the platform not only accelerates procurement but does so **safely and in accordance with legal and ethical standards**.

Reliability, SLOs, and Event Replay

Hospitality.Watr.Exchange is intended to be a mission-critical system for its users, so high reliability and defined Service Level Objectives (SLOs) are essential.

Availability SLO: We target an uptime of e.g. 99.9% or higher for the core services, meaning minimal downtime per month. Using Azure’s redundant infrastructure, each component (API gateway, event processing, database) is deployed in a highly available way (multiple instances, across availability zones where possible). For example, Azure Service Bus and Event Grid are highly available by default, and we run at least two instances of each microservice (so one can handle traffic if the other fails). The system is designed with **failover** in mind – if one processing function fails, messages remain in the queue and another instance picks them up.

Scalability and Performance SLOs: We also set targets for performance – e.g., the ability to process X orders per minute, or to complete an auction end-to-end within a certain time after closing (say, auction results published within 1 second of auction end). The event-driven model allows concurrency, but we will test and tune for peak loads. If the exchange is used by thousands of restaurants, it could generate high event volumes (imagine lunch time ordering spikes); auto-scaling rules on Functions and container instances will handle that. We also consider using caching/CDN for any static reference data (like product catalog queries) to offload the core.

Durability and Replay: One of the biggest advantages of our Persist-and-Publish model is the ability to **replay events**. This is crucial for both reliability and debugging: - If a bug in the Auction Service caused it to mis-evaluate a winner, developers can fix the code and replay the relevant sequence of events (bids, etc.) from the event store to recompute outcomes – useful for post-mortem or even to “re-run” an auction simulation if needed. - If a new service comes online (say we add a new Analytics service later), it can read the backlog of events from the store to build historical data. - In case of partial failures, e.g., the AP Service was down for an hour, all invoice events during that hour are safely stored and not lost – once it’s up, it can retrieve them (either via Service Bus dead-letter queues or by querying the event store for events it missed based on timestamp). - We implement **idempotence** in event handlers: if the same event comes twice, the handler can detect it was already processed (maybe by checking a process log or using the event ID as a key in a handled set). This prevents duplicate processing which is a common concern in at-least-once delivery systems.

Error Handling: Each microservice will have error queues or dead-letter topics. If a message/event cannot be processed (maybe schema validation fails or a business rule exception), that event can be moved to a dead-letter channel and an alert is raised. Support engineers can examine it, fix the data or code, and then re-inject or replay that event. This prevents one poison message from blocking the pipeline.

Consistency: Because we are event-driven, we embrace eventual consistency. There might be slight delays between an order event and all services catching up (maybe milliseconds or a few seconds under load). This is acceptable for our domain – a slight delay for an invoice to show up after an order is placed, for example, is fine. But we ensure eventual consistency: given a short time, all projections will converge to the correct state from the single source of truth (the event log). We also design idempotent, consistent workflows: e.g., if two winning bids events somehow got published due to a glitch, the Sourcing Service might detect that an order already exists for that auction and ignore the duplicate.

Backup and Recovery: All persistent data stores (Cosmos DB, SQL, Blob) have backups or point-in-time restore enabled. The event store could be geo-replicated for disaster recovery. In worst-case scenarios (data center outage), we can restore from backups or switch to secondary region. The microservices themselves can be redeployed from code at any time; stateless design ensures we can recover by replaying events into a fresh instance.

Testing for Reliability: We will conduct **chaos testing** and load testing. Azure's Chaos Studio or custom scripts can simulate outages of services to ensure the system self-heals or degrades gracefully. For example, if the Compliance Service is down, does that halt all orders? Ideally not – maybe orders proceed but get flagged for later review. Or the system might have a fallback to allow trading to continue with minimal checks and alert admins if compliance checks were skipped. These decisions will be part of SLO-driven design – maybe we accept a short downtime of compliance service if needed, but not of the core order intake.

Monitoring SLOs: We define clear SLO metrics like: - **Order processing latency:** 99th percentile of time from order submission to supplier acknowledgment event is < 5 seconds (for instant buys). - **Auction event latency:** Bids are propagated to all participants in < 1 second. - **System throughput:** support at least 100 events/sec with current architecture, scalable to 1000+ events/sec with additional function instances.

If these SLOs are threatened (say we hit 80% of capacity), Azure alerts will trigger scale-out or at least notify DevOps to add capacity. The cloud design allows dynamic scaling to meet these objectives as long as cost is managed.

In summary, reliability is built into the exchange via redundancy, event sourcing (replay), and careful isolation of failures. Users should experience a dependable platform, and even in the rare case of an issue, the system can recover state from its durable log. This ensures trust – if a restaurant places an order, they won't have to wonder "did it go through?" because the system guarantees that either it completes or it's recorded for follow-up (no silent failures). Our goal is that the exchange can run as continuously and reliably as Toast's POS systems – it becomes an integral piece of the operational puzzle that cannot afford to go down during dinner rush.

Ranking Logic and Analytics

The platform not only executes transactions but also **optimizes and learns** from them. The Ranking and Analytics aspects provide the intelligence layer on top of raw exchange operations.

Ranking Logic: In many cases the platform must decide an outcome that is not purely first-come-first-served. As mentioned, auctions might consider factors beyond price; also the platform might rank suppliers (for example, if multiple suppliers post standing offers at the same price, which one do we show on top to buyers?). This is where a **scoring function** comes in. We define a scoring model that can weigh multiple attributes: - Price (lower is better for buyer-oriented ranking). - Quality metrics (perhaps derived from past ratings or fulfillment performance). - Lead time or distance (a local supplier might be preferred for speed or freshness). - Reliability (supplier fill rate history – if a supplier often cancels, they might be ranked lower). - Buyer's preferences (maybe a buyer favors vendors who are sustainable or have certain certifications).

In an automated Instant Buy scenario, if multiple offers satisfy a request, the Sourcing Service can use a ranking function to auto-select the best offer (if the buyer opted for an "auto-choice best of X suppliers" mode instead of picking manually). Similarly, for auctions, if we allowed multi-attribute bids (rare in reverse auctions, but possibly if quality can vary), a **scoring auction** approach would standardize bids into a score[22]. However, likely our auctions are purely price on a fixed spec, so ranking there mostly means time priority (who accepted first at a price).

Another area for ranking is **barter matching**: if multiple cycles are possible, which one to execute first? The Barter service might score cycles by total utility – e.g., a cycle that trades larger volumes or involves high-need items could be prioritized. This could align with maximizing overall benefit or number of participants served.

The Ranking Service could implement algorithms like weighted sum or even machine learning models to predict best matches. For launch, simple rule-based scoring is sufficient, but as data accrues, we might train models on successful trades vs failed negotiations to refine ranking.

Analytics and Insights: With all transactions flowing through, the exchange accumulates valuable data: - Pricing trends over time for each commodity. - Supplier performance stats (average delivery times, bid win rates, average prices offered). - Buyer behavior (what times of year they purchase certain items, how often they resort to auction or barter). - Marketplace health (number of active offers, ratio of auctions that closed successfully vs no bids, etc.).

We will build dashboards and reports for internal use and for customers: - **For internal/Toast:** analytics help tweak the platform. E.g., if auctions for produce always fail to attract bidders, maybe we need more produce suppliers or adjust the auction format. If barter usage is low, maybe it needs more education or certain incentives. - **For buyers:** we can show analytics like "You saved \$X this quarter by using the exchange" or "Supplier Y provides you the fastest deliveries". These value reports strengthen customer satisfaction. - **For suppliers:** insights like market demand forecasting ("restaurants in your area are searching for local beef – consider listing an offer") or pricing guidance ("your bid prices are slightly above market average, consider adjusting to win more auctions").

Technically, our Analytics pipeline can use Azure Stream Analytics or Azure Databricks to consume the event stream and update aggregate metrics in near-real-time. Some metrics might be computed daily (end-of-day batch jobs on the event store). We will also integrate with Toast's analytics if relevant (perhaps combining sales data with purchase data to analyze margin).

Machine Learning Potential: In later phases, advanced analytics could enable features like **predictive procurement** (flagging a restaurant: "you will likely run out of item X in 3 days, initiate sourcing now") or **dynamic pricing** suggestions for suppliers (like how ride-sharing or stock markets work, price could vary with demand – we could eventually automate Instant Buy prices based on supply/demand using ML models if desired).

Ranking for UX: The platform's UI might list suppliers or offers in some order – likely that order is determined by a combination of price and supplier rating. We ensure this logic is transparent and fair (maybe rotating among equals to give everyone a chance, to avoid accusations of favoritism).

In summary, the exchange doesn't just passively connect buyers and sellers; it **adds intelligence**. By ranking offers and analyzing outcomes, it continuously improves the efficiency of matches. This leads to a virtuous cycle: better matches -> more success -> more users -> more data -> better algorithms. For Toast, this analytics capability can become a significant value-add and even a product of its own (e.g., selling industry reports or supplier scorecards). The key is that our architecture (with all data events centrally available) makes it straightforward to build such analytics without impacting transaction processing.

Pilot Plan and Implementation Roadmap

Launching a platform of this scope benefits from a phased approach. We propose the following roadmap with pilot phases to validate and refine the system:

Phase 1: Prototype and Internal Testing (Months 1-3) - Build a **functional prototype** of the core exchange with a simplified scope: maybe just Instant Buy and a basic Auction mechanism. Use a small set of test data (a few products, a fake buyer, and a couple of supplier accounts). - Run it in a controlled environment. Simulate transactions to test the ATP flow, event handling, and integration. For example, generate a few 850 orders and ensure they become events and lead to orders in the system. - Internally at Toast, have a few employees act as buyers/suppliers in a sandbox to get usability feedback. - Validate the integration with Toast's POS/inventory system on a limited scale: e.g., have one restaurant location's manager use the system to place a dummy order for something.

Phase 2: Limited Pilot with Friendly Customers (Months 4-6) - Recruit a small number of **friendly pilot participants**: e.g., 2 restaurant groups (maybe a total of 5-10 restaurant locations) and 2-3 suppliers they work with regularly. Onboard them to the platform. - Focus initially on **Instant Buy and Auctions**, since barter is more complex and might need more network breadth. The pilot could involve these restaurants placing some of their real orders through the platform to those suppliers. The suppliers will list offers or engage in auctions for that pilot. - Set up EDI integration for one supplier if needed, to test the EDI workflows end-to-end (maybe one supplier continues using their ERP, so we exchange 850/855/856/810 with them via our EDI module). - Monitor performance and collect feedback: Are orders processing correctly? Do auctions result in satisfactory outcomes? Use this to refine the user interface, fix any reliability bugs, and maybe tune the auction parameters (e.g., price drop interval). - We'll also measure pilot KPIs: e.g., did the restaurants save money in these few orders? Was the process faster or slower than their usual way? This helps build a case study.

Phase 3: Barter Pilot and Full Feature Rollout (Months 6-9) - Introduce the **Barter rail** in a controlled scenario. Perhaps find a cluster of 3-4 restaurants under common management that might try bartering among themselves or with a known partner. For example, two hotels in a city trade excess inventory. We can simulate or encourage a specific barter deal to test the engine. - If direct multi-party barter is hard to arrange at first, consider a simpler case: bilateral barter (which is just a swap) as a subset to ensure the system works. Then extend to 3-party. - This phase will involve a lot of testing of the GCBE algorithm's matching logic. We might generate artificial offers to ensure the matcher finds cycles, verifying the lock/escrow mechanism and that the outcomes are fair and balanced. - Also, expand the auction usage to multi-lot scenarios to test having multiple winners, and ensure the Auction Service and subsequent order splitting works correctly. - By the end of this phase, all three rails are functioning. We ensure documentation is in place (user guides for how to create auctions, how to list barter offers, etc.), as well as internal runbooks for operations (how to monitor a stuck event, etc.).

Phase 4: Scaling Up – Beta Launch to More Customers (Months 9-12) - Open up the platform to a broader segment of Toast's customer base as a **beta**. For example, allow any restaurant in one region to sign up and try it with a limited number of suppliers (we might onboard a set of suppliers for common categories – food distributors, cleaning supply companies, etc.). - We will likely do region by region or category by category to manage growth. Perhaps start in one city or state to ensure focus. Also possibly choose one procurement category (like produce) to avoid complexity, then expand to others. - Use incentives to encourage adoption: e.g., waive transaction fees during beta, or offer some credit for trying an auction or barter trade. Network effect is important – we need enough suppliers in a category for buyers to get responses. So a concerted recruitment of suppliers happens here (possibly leveraging Toast's existing supplier relationships or partners). - During beta, closely monitor system performance under real load. This is where our SLOs are tested. We'll refine scaling parameters, optimize any slow queries (for example, if the event store gets large, ensure partitioning is in place). - Gather user feedback and success stories. If beta goes well, marketing can prepare testimonials: e.g., "Restaurant X saved \$500 in their first week using Watr.Exchange".

Phase 5: General Availability (after Month 12) - Incorporate all feedback from beta, finalize the pricing model for monetization (transaction fees, etc.). Harden the system for production (do a security audit/pen-test, ensure compliance docs are up to date). - Officially launch to all Toast customers. This likely involves a marketing push and making the feature easily accessible in the Toast dashboard. - Continue to add minor enhancements that were deferred: for example, more advanced analytics dashboards, maybe a mobile app interface for on-the-go auction monitoring, integration with inventory forecasting, etc.

Future Roadmap Ideas: - Explore integration with payment processing to offer a *guaranteed payment* or Toast-financed credit line for buyers, adding value on financial side. - Consider introducing **FEMM (Fair Entropy Market Maker)** concepts to complement GCBE, as hinted by research[23][24]. For example, implementing dynamic pricing algorithms that adjust offers based on reducing "friction" but preventing speculation. This could be an advanced feature where the exchange uses AI to suggest price adjustments to stabilize supply/demand. - Leverage blockchain or distributed ledger for an **audit trail** of barter trades or to potentially tokenize goods for easier multi-party exchange (this is experimental, only if it adds trust or efficiency). - Expand beyond hospitality: if successful, the platform concept could extend to other domains (maybe retail or other SMB procurement) as a general Watr.Exchange network.

Throughout all these phases, we maintain a tight loop between business feedback and technical iteration. The whitepaper provides the blueprint, but real-world usage will guide fine-tuning. By starting small and scaling thoughtfully, we mitigate risk and prove value at each step, ensuring that by the time we go broad, we have a stable, impactful product.

Appendices

Appendix A: Canonical Event Schemas

Below are example schemas for some key events in JSON format (using an informal schema notation for brevity):

- **PurchaseOrderRequested Event Schema:**
- type: `com.hospitalitywatr.order.requested`
- source: `/orgs/{buyerId}/orders`
- `data`:
    - `orderId` (string, GUID)
    - `buyer` (object with `orgId`, name)
    - `supplier` (object with `orgId`, name – may be null if not assigned yet in auction context)
    - `items` (array of line items, each with `productId`, `description`, `quantity`, `unitPrice`)
    - `totalPrice` (number)
    - `exchangeMechanism` (string enum: "InstantBuy" | "Auction" | "Barter")
    - `relatedAuctionId` (if came from an auction, reference ID)
    - `timestamp` (string, ISO datetime)
- **AuctionClosed Event Schema:**
- type: `com.hospitalitywatr.auction.closed`
- source: `/auctions/{auctionId}`
- `data`:
    - `auctionId` (string)
    - `winningBids` (array of objects):
    - each with `supplierId`, `acceptedPrice`, `quantityAwarded`
    - `clearingPrice` (number – if single price for all winners, e.g., in Dutch auction all winners pay the price at which last unit was filled)
    - `unfilledQuantity` (number, if any portion went unfilled)
    - `status` (string, e.g., "Completed", "Partial", "NoBids")
    - `timestamp`
- **ExchangeCycleFormed Event Schema:**
- type: `com.hospitalitywatr.barter.cycleformed`
- source: `/bartercycles/{cycleId}`
- `data`:
    - `cycleId` (string)

  - `participants`: array of org IDs involved
  - `transfers`: array of transfer objects, each describing one directed transfer:
  - `fromOrg`, `toOrg`, `productId`, `quantity`
  - Example transfers might be:

    ```
    [
      {"fromOrg": "A", "toOrg": "B", "product": "Wine", "quantity": 5
    0},
      {"fromOrg": "B", "toOrg": "C", "product": "OliveOil", "quantity
    ": 20},
      {"fromOrg": "C", "toOrg": "A", "product": "Flour", "quantity":
    100}
    ]
    ```

    which indicates A->B (50 units Wine), B->C (20 units OliveOil), C->A (100 units Flour).
  - `timestamp`
- **InvoiceIssued Event Schema:**

- `type`: `com.hospitalitywatr.invoice.issued`
- `source`: `/invoices/{invoiceId}`
- `data`:
  - `invoiceId` (string)
  - `orderId` (string, link to order)
  - `supplierId`, `buyerId`
  - `amount` (number, total invoice amount)
  - `currency` (string, e.g., "USD")
  - `dueDate` (date)
  - `lines`: array of { `productId`, `quantity`, `price` } for detail (could be optional if not needed)
  - `timestamp`

These schemas would be defined in more detail (possibly using JSON Schema definitions) and versioned. All events include the CloudEvents context properties as well (specversion, id, source, type, time, etc.). In code, we may use classes or structs to represent them for validation.

Appendix B: Auction Clearing Pseudocode

For a reverse Dutch auction (descending clock, possibly multi-unit), here is a more detailed pseudocode for the auction process:

```
# Assume auction parameters given:
current_price = start_price
auction_end_time = now + duration
price_decrement = decrement_step
next_tick = now + tick_interval
quantity_remaining = total_quantity
winners = []  # list to store winning allocations

while now < auction_end_time and quantity_remaining > 0:
    wait_until(min(next_tick, auction_end_time))  # wait until next price dro
p or end
    # Check for any acceptances during the last interval
    for bid in get_new_acceptances():  # any supplier accepted the current_pr
ice
        if quantity_remaining <= 0:
            break
        accept_qty = min(bid.quantity_offered, quantity_remaining)
        winners.append({ "supplier": bid.supplier_id,
                         "price": current_price,
                         "quantity": accept_qty })
        quantity_remaining -= accept_qty
        # Lock the supplier out from further bidding and reduce needed quanti
ty
        mark_supplier_won(bid.supplier_id)
        if quantity_remaining <= 0:
            break
    if quantity_remaining <= 0:
        break
    # No acceptance in this interval, or still quantity left, drop price
    current_price = max(current_price - price_decrement, reserve_price)
    publish_price_drop(current_price)  # notify bidders of new price
    next_tick = now + tick_interval

# Auction ends: either time up or quantity filled
if quantity_remaining > 0:
    # time ran out without full allocation
    # No more price drops; possibly partial fill or no fill
    pass  # winners list may be partially filled
else:
    # full quantity allocated (or as much as available among bidders)
    pass

publish_auction_closed(winners, clearing_price=current_price, unfilled=quanti
ty_remaining)
```

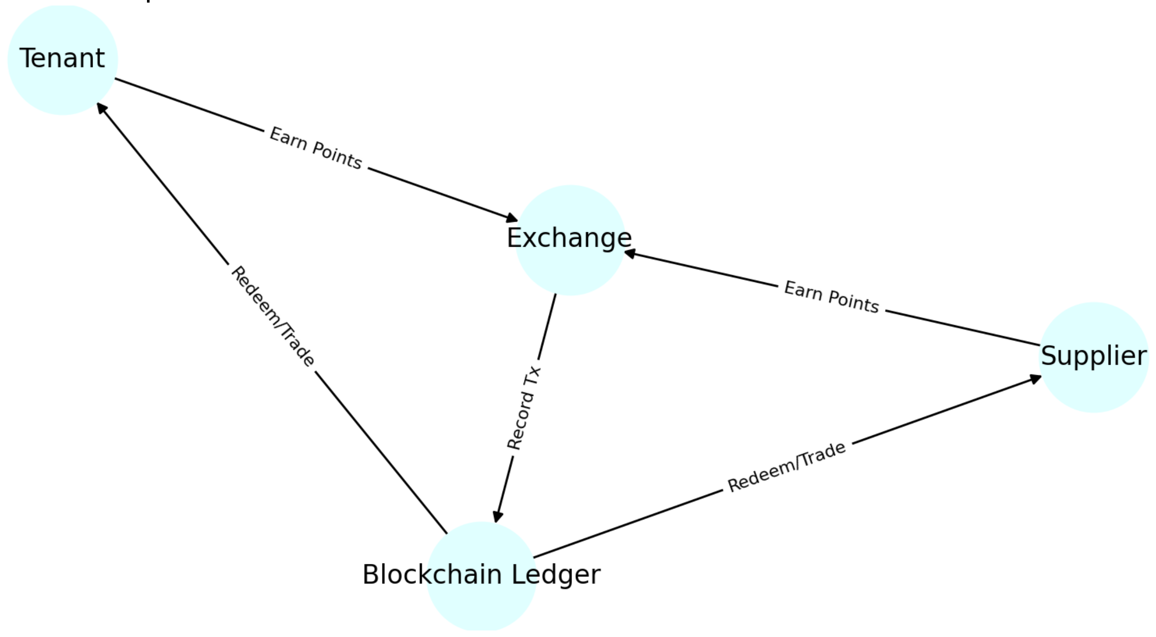
In words: - The price starts at a ceiling. At each tick, if no one has taken the offer at that price (for the needed quantity), the price is lowered. - The first moment a supplier accepts, they win at that price for whatever quantity they committed. If that fulfills the whole quantity, auction ends immediately. If it fulfills part, that quantity is removed from the demand and the auction continues for the remainder (possibly with the price continuing to drop to attract others for the rest). - If multiple suppliers accept *simultaneously* at a price drop (e.g., the instant price went from \$10 to \$9, two suppliers jump in at \$9), then we allocate on a first-come basis. Our system will likely timestamp each acceptance and sort them. Or we might allocate pro-rata if fairness required, but typically first-come in Dutch. - If the clock runs out (reaches a predefined minimum or just time expires) and some quantity is still unfilled, then the auction closes with partial success. The buyer can then choose to manually negotiate for the rest or re-run, etc.

This pseudocode would be implemented by the Auction Service, which also concurrently listens for acceptances. In a distributed system, acceptances could be events from suppliers (e.g., hitting an API "AcceptOffer" which triggers a BidPlaced event). The Auction Service might maintain an in-memory state for each live auction to handle this logic, ensuring low-latency response.

Appendix C: GCBE Matcher Diagram and Explanation

The GCBE barter matching process can be visualized with a flowchart or state diagram:

(Refer to Figure 3 earlier for an example cycle.) The core steps: 1. **Collect Offers:** Input is the set of active barter offers (asks). Each offer may have multiple legs (sub-offers) if it involves multiple



commodities. 2. **Build Graph:** Create a directed graph where each offer (leg) is a node. Draw a directed edge from Offer X to Offer Y if the commodity that X is offering can satisfy the commodity that Y wants, and their ratio constraints can align. Essentially, X->Y means "X's output can feed Y's input." 3. **Search for Cycles:** Look for a cycle in this graph that returns to the start. Various graph algorithms (like a depth-first search with path tracking up to a certain depth, or using network flow algorithms for cycles) can be used. The `MaxLegDepth` on offers might restrict how deep we go[25] (to limit complexity). 4. **Validate Cycle Quantities:** If a cycle is found (e.g., A->B->C->A), ensure that each can exchange in quantities that meet each minimum requirement. Often, you determine a feasible exchange quantity by finding the minimum capacity along the cycle. For example, if A has 100 units to give, B wants at least 80 of that, C can only supply enough to give A equivalent of 50 – the cycle might be limited to 50 units trade based on the bottleneck. 5. **Execute Cycle:** Lock all involved offers (so they aren't matched elsewhere). Create orders/shipments as needed for each pair in the cycle. Remove or decrement the available quantities from these offers (e.g., if not fully exhausted). 6. **Loop:** Continue to find cycles until no more possible or until a pause to wait for new offers.

A pseudo-code for matching might look like:

```
for each offer in offers:
  if offer is already satisfied, skip
  depth-first search offer to find a path back to itself:
    keep track of visited offers
    for each neighbor (edge) from current:
       if neighbor == start_offer:
           success: cycle found
       else if neighbor not visited:
           dfs(neighbor)
```

During the DFS, also accumulate the ratio of exchange to compute how much could flow. This can get complex with multi-leg, but essentially you translate quantities through ratios along the path.

The **transformation** aspect means some edges in the graph could represent not a direct want-have match but a conversion. For example, an edge might represent "Offer X's commodity can be converted by Supplier Y to what Offer Z needs." We can model that by inserting a node for the transformation that has edges from X to Transform node, and from Transform node to Z. The transform node would have its own constraints (e.g., a conversion rate or fee).

The diagram might also incorporate the escrow: after cycle found, state moves to "Escrow stage" where each participant is asked to confirm availability. Only after all confirm does it move to "Execute trade" and emit the cycle event and orders.

Overall, the matcher ensures **multi-lateral trade closure**, which as cited earlier, drives speculative value to near zero by ensuring everyone gets real goods[6]. This engine has to run either continuously or at intervals. We might run it every time a new offer comes in (since that could enable a new cycle) or after a batch of offers.

Appendix D: BPMN Process Diagrams

To better understand process flows, BPMN (Business Process Model and Notation) diagrams were used. Figure 1 in the main document already showed a high-level process with swimlanes for a Hospitality Org, the Exchange, Supplier, and Manufacturer. Additional BPMN diagrams can detail specific scenarios:

- **Order to Payment Flow (P2P cycle):** showing the steps from order placement, through acknowledgment, shipping, receiving, invoicing, and payment, with both buyer and supplier swimlanes and the exchange in between handling communications.
- **Reverse Auction Process:** a BPMN diagram could illustrate how a buyer starts an auction (event), suppliers get notifications, they place bids (or accept price drops) in their swimlane, and the auction closes with one or more winners, then proceeding to order creation.
- **Barter Cycle Negotiation:** if we consider a semi-automated negotiation, a BPMN could show participants posting offers, the system suggesting a cycle, and participants confirming, etc.

For brevity, we include one detailed BPMN diagram here as an example (Figure 1 in the main text). Additional diagrams can be produced as needed for internal documentation or training materials, ensuring all stakeholders (even non-technical) grasp how the processes will change with Hospitality.Watr.Exchange in place.

In all, the combination of narrative, technical architecture, and visual diagrams in this whitepaper provides both the strategic rationale and the implementation blueprint for Hospitality.Watr.Exchange. By following this guide, Toast's leadership can envision how the exchange will create value and the engineering teams have a clear path to build and roll out this innovative platform in the hospitality market.
