LOVELY PROFESSIONAL UNIVERSITY

CSB205: SYSTEM DESIGN

Major Project

Topic: Multi-Tenant E-Commerce Marketplace with Search & Recommendations

Team Names: Perineni Lakshith, Rachana Reddy, Jashwanth Raparthi, Praveen Kumar, Rakesh Reddy
Team Reg.No's: 12401218, 12400989, 12406316, 12405070, 12406880

Section: K24BX

1. Requirements Pack

1.1 Stakeholders

A successful marketplace requires satisfying diverse needs across three main user groups and the core technology team.

Stakeholder Group	Core Needs / Domain Focus
Sellers (Tenants)	Storefront customization, inventory management, order processing, sales analytics, billing.
Buyers (End Users)	Personalized product search, seamless cart/checkout, order tracking, reviews, secure payments.
Platform Administrators	Tenant oversight, policy enforcement, platform-wide configurations, system health monitoring.
Development/DevOps Team	CI/CD automation, easy scaling, robust monitoring, fault isolation, cost efficiency.

1.2 User Stories (Feature Context)

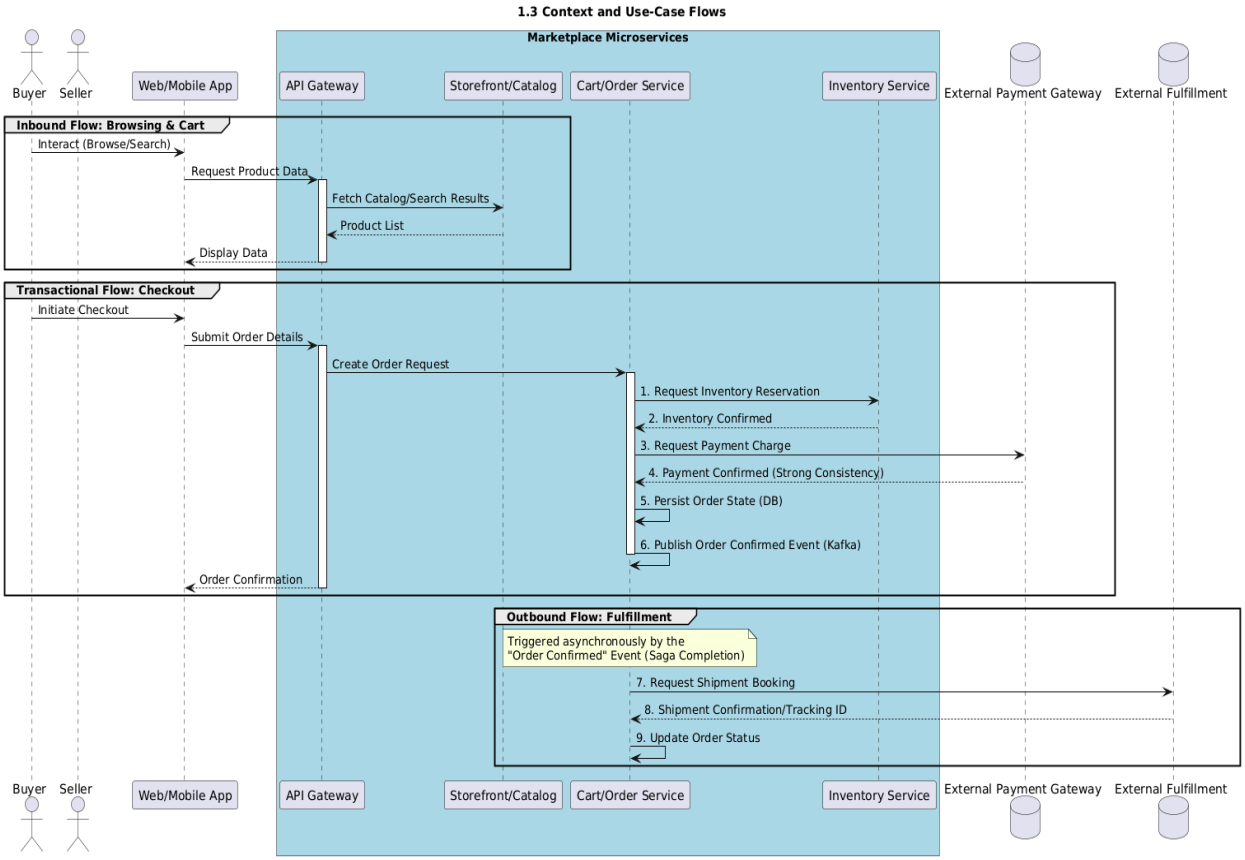
Persona	User Story	Core Feature Implied
Seller	"As a Seller, I want to **upload bulk inventory via CSV** so that I can quickly populate my catalog."	Catalog & Inventory Service, Data Ingestion Pipeline
Buyer	"As a Buyer, I want **search results to prioritize items based on my browsing history** so that I find relevant products faster."	Search Service, Recommendation Engine
Admin	"As an Admin, I want to **disable a tenant's storefront immediately** if	Tenant Service, Identity & Access Control

	they violate policy."	

1.3 Context and Use-Case Diagrams

The system operates as a central nervous system coordinating interactions between end-users, tenants, and various external systems.

- **Inbound Flow:** Buyers interact via Web/Mobile App -> API Gateway -> Storefront Service (for browsing) or Cart/Order Service (for transactions).
- **Outbound Flow:** Order Service -> Payment Gateway (e.g., Stripe, PayPal) -> Shipping/Fulfillment Provider (e.g., FedEx, UPS).



1.4 Constraints and Assumptions

Category	Constraint	Assumption
Non-Functional	Elastic scale for traffic spikes, strong consistency for payments/orders, strict PII protection.	APIs (Payments, Fulfillment) are generally reliable and available.
Technical	Architecture must use Domain-Driven Design (DDD) principles and leverage event choreography.	The team is proficient in multiple languages (Go/Java) and Kubernetes.
Regulatory	PCI-aware design for payment handling, compliance with data privacy laws (GDPR/CCPA).	PCI scope is minimized via external payment processors (tokenization).

---

2. Architecture

2.1 Trade-off Analysis of Styles

The **Microservices Architecture** was chosen to meet the non-functional goals of elastic scale, fault isolation, and technology heterogeneity required by a modern, high-traffic marketplace.

Style	Rationale for Rejection/Acceptance	Trade-offs Accepted
Monolith	**Rejected.** Fails to meet elastic scale requirement (e.g., scaling Search independently). Leads to a single point of failure.	N/A
Microservices	**Selected.** Allows independent deployment,	**Accepted:** Operational complexity, distributed

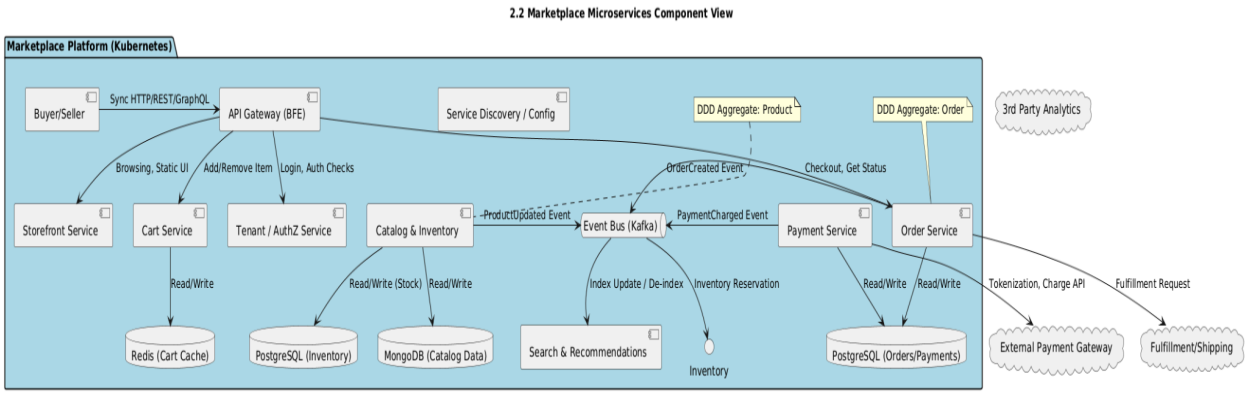
	technology specialization (e.g., Go for performance, Java for complex logic), and fault boundaries.	transaction handling, eventual consistency.
Serverless	**Hybrid Use.** Suitable for asynchronous tasks (e.g., image resizing, report generation) but poor for long-running transactional Sagas.	**Accepted:** Potential cold start latency for non-critical paths.

2.2 Component Diagram

The system is divided into core domains, communicating asynchronously via an Event Bus (Kafka) and synchronously via the API Gateway.

Key Services/Domains and DDD Contexts:

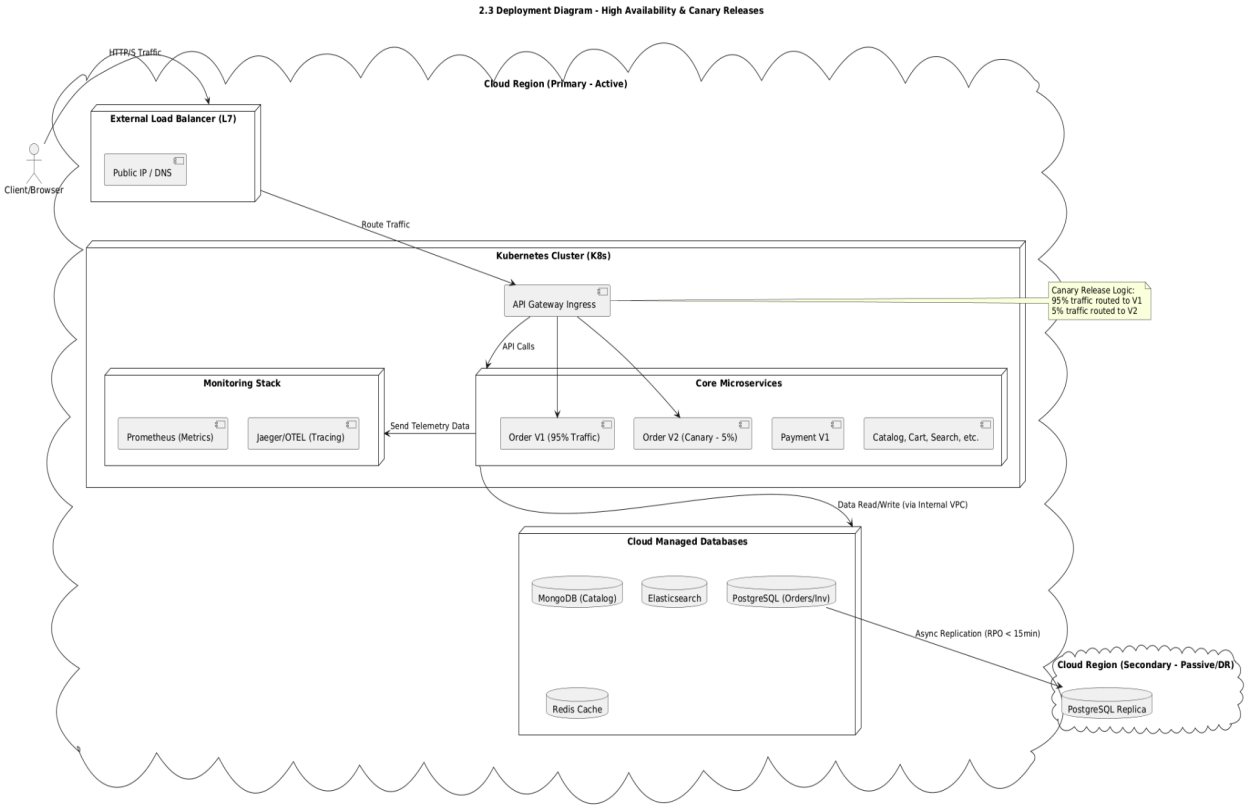
- **API Gateway (BFE):** Acts as a Backend-for-Frontend, responsible for request aggregation (e.g., fetching product, reviews, and inventory in one call for a Product Detail Page), authentication, and rate limiting.
- **Storefront:** Buyer UI/UX, static content delivery.
- **Catalog & Inventory:** Product details (NoSQL), stock levels (Relational). **DDD Detail:** The Product is the Aggregate Root here.
- **Search & Recommendations:** Elasticsearch/AI components for personalization.
- **Cart:** Manages buyer sessions, uses Redis cache.
- **Order:** Manages the order lifecycle (DDD Aggregate Root). All state changes are validated within this boundary.
- **Payment:** Handles tokenization, reconciliation, and provider communication.
- **Tenant/AuthZ:** Manages seller accounts, roles, and permissions.



2.3 Deployment Diagram

The architecture is deployed on Kubernetes (K8s) for orchestration, enabling high availability and controlled releases.

- **Platform:** Kubernetes (K8s) for container orchestration, providing self-healing and service discovery.
- **Load Balancing:** External Load Balancer (L7) routes traffic to the API Gateway.
- **Deployment Strategy (Canary Releases):** For critical services (e.g., Payment, Order), K8s is configured to use **Canary Releases**. New versions are initially rolled out to a small subset (e.g., 5%) of production traffic, monitored extensively for performance and error rates via Prometheus/Jaeger, before proceeding with a full deployment roll-out. This minimizes blast radius during software updates.
- **Database:** Cloud-managed services (e.g., AlloyDB/PostgreSQL, Managed MongoDB/CosmosDB, Elasticsearch Cluster) to reduce operational overhead.



3. Design Patterns

The architecture relies heavily on domain-specific patterns to manage complexity and distributed state.

Pattern	Type	Implementation Detail	Anti-Patterns Avoided
Saga	Behavioral	**Order Lifecycle:** Uses event choreography (e.g., OrderCreated -> InventoryReserved -> PaymentCharged). Guarantees eventual consistency for distributed transactions.	Two-Phase Commit (2PC) in a distributed system (too slow/complex).
Outbox	Behavioral	Ensures that database updates (e.g., Order status change) and the publishing of corresponding events (e.g., OrderConfirmed) are atomic via a dedicated outbox table.	Dual write problem (inconsistent state between DB and Event Bus).
Factory	Creational	**Payment Provider Factory:** Decouples the Payment Service from concrete payment gateways (Stripe, PayPal) by creating provider-specific	Excessive conditional logic (if/else hell) for external system integration.

		classes via a common interface.	
Adapter	Structural	**Fulfillment Adapter:** Standardizes the interface for heterogeneous external fulfillment services (FedEx, UPS) so the Order Service doesn't rely on vendor-specific logic.	Direct reliance on external vendor APIs, making vendor switching difficult.
Facade	Structural	**Search Facade:** Provides a simple search(query) interface to the Storefront, hiding the complexity of the Elasticsearch query language, ranking, and personalization logic.	Clients needing deep knowledge of the underlying search technology.

---

4. Database Design

4.1 Polyglot Persistence Rationale

Choosing the right data store per service optimizes performance and consistency for specific domain needs.

- **Relational (PostgreSQL/AlloyDB):** Used for **Orders, Payments, User Accounts**. *Rationale:* Non-negotiable strong consistency (ACID), transactional integrity, and complex financial reporting where joins are necessary.
- **Document Store (MongoDB):** Used for **Product Catalog**. *Rationale:* Flexible schema to support diverse product attributes (e.g., clothes vs. electronics) across different sellers (tenants) without schema migrations.

- **Search Index (Elasticsearch):** Used for **Product Discovery, Recommendations**. *Rationale:* Optimized for full-text search, fuzzy matching, and fast aggregation (faceting) required by the Buyer experience.
- **Cache (Redis):** Used for **Cart Data, Session State, Popular Product Lookup**. *Rationale:* High-speed, low-latency key-value store, supporting ephemeral state.

4.2 Entity-Relationship Diagram (ERD) / Schema Overview

4.3 Normalization/Denormalization Rationale

- **Normalization (3NF):** Applied strictly to **Orders/Payments** tables within Postgres. Ensures data integrity, prevents update anomalies, and supports complex joins for financial audits.
- **Denormalization:** Applied in the **Catalog Service** (embedding product features, reviews summaries) and heavily in the **Search Index** (flattening product data for fast read access). *Rationale:* Optimized for read-heavy operations, trading write complexity for massive read speed improvements.

4.4 Indexing, Partitioning/Sharding Plan

- **Indexing:** All relational tables must be indexed on the composite key (tenant_id, primary_key) for efficient multi-tenant query isolation. Secondary indexes on (tenant_id, created_at) are critical for tenant-scoped reporting.
- **Partitioning/Sharding (Horizontal Scaling):**
    - **Strategy:** Shard data primarily by **tenant_id**. This achieves **Tenant Isolation** and distributes the load from large tenants across different database instances, minimizing the "noisy neighbor" effect.
    - **High-Volume Tables (Orders):** Further partitioning by date range (e.g., monthly) within a shard can improve archive and query performance.
- **Data Synchronization (CDC):** Data is synchronized between the transactional (Postgres/Mongo) and search (Elasticsearch) stores using the **Change Data Capture (CDC)** mechanism. The Outbox pattern ensures every relevant database change generates an event, which is then consumed by the Search Service to maintain the up-to-date index, guaranteeing read-model eventual consistency.

---

5. Performance & Scale

5.1 Caching (In-Memory/CDN)

Cache Layer	Technology	Data Stored	Strategy
L3 (CDN)	Cloudflare/Akamai	Static assets, Seller logos, Product images.	Edge caching, Geo-distribution for global user base.
L2 (Distributed)	Redis Cluster	Shopping Carts, Session State, Personalized Recommendations, Hot Product IDs.	TTL-based expiration, High-speed lookup, handles sudden spikes in read traffic.
L1 (In-Memory)	Service Local Cache	Feature flags, Configuration values, Tenant account details (frequently accessed).	Short TTL, small data sets, eliminates network hop for critical config.

5.2 Load Balancing Strategy

- **Layer 7 (Application Load Balancer - ALB):** Used at the API Gateway level. It can inspect HTTP headers and hostnames, allowing for intelligent routing based on tenant subdomain or path.
- **Strategy:** Primarily **Least Connections** to ensure new requests are routed to the least-busy service instance, maximizing utilization and minimizing individual instance load.

5.3 Replication

- **Database:** Asynchronous streaming replication (read replicas) for all relational stores (Postgres) to handle heavy read traffic (e.g., reports, catalog views) and ensure database resilience.
- **Services:** All microservices run with a minimum of 3 replicas across multiple Availability Zones (AZs) in K8s to withstand zone-level failures.

5.4 Backpressure

- **Mechanism:** Implemented at the API Gateway using the **Token Bucket Algorithm**.
- **Function:** Enforces **rate limiting** policies configured per tenant_id and global service limits. This prevents a surge of traffic from one misbehaving tenant (a "noisy neighbor") from degrading service quality for all others, ensuring system stability.

---

6. Security & Reliability

6.1 Threat Model (Core Defenses)

Threat Category	Example Threat	Primary Defense Mechanism
Access Control	Unauthorized tenant accessing another tenant's data.	**Row-Level Security (RLS)** in databases, enforced checks in services using tenant_id.
Denial of Service (DoS)	Traffic flooding the search endpoint.	**WAF** (Web Application Firewall) for L7 attacks, and **Rate Limiting** at the API Gateway.
Data Exposure	Payment details interception.	**PCI-aware design** (tokenization), TLS 1.3 encryption end-to-end, Secret Rotation (using KMS/Vault).

6.2 Authentication (AuthN) and Authorization (AuthZ)

- **AuthN:** Handled by a centralized **Identity Provider (IdP)** (e.g., Keycloak). Uses **OIDC** and **JWT (JSON Web Tokens)** for stateless session management.
- **AuthZ: Role-Based Access Control (RBAC)** enforced at two critical levels:
  1. **API Gateway:** Validates JWT structure and scopes/roles before routing.
  2. **Service Level:** Each service validates the user's tenant_id and role against the

resource being accessed.

- **Tenant Isolation via RLS:** Row-Level Security (RLS) is paramount for multi-tenancy. In PostgreSQL, RLS policies are enforced to ensure that a query originating from a service authenticated with tenant_id=X can *only* retrieve or modify rows where the tenant_id column equals X. This provides a fundamental security layer against cross-tenant data access attempts.

6.3 OWASP Top 10 Defenses

- **A03: Injection:** Use **Parameterized Queries/ORMs** (e.g., Hibernate, GORM) instead of string concatenation for database access to eliminate SQL Injection risk.
- **A04: Insecure Design/Broken Access Control:** Enforce multi-layer access checks, especially using the tenant_id filter on all read/write operations.
- **A05: Security Misconfiguration:** Use infrastructure-as-code (IaC) and secret rotation services to manage configuration securely.

6.4 Resilience Patterns

- **Circuit Breakers:** Implemented using libraries (e.g., Hystrix, Resilience4j) on every synchronous inter-service call. If the Review Service fails, the Order Service opens the circuit, and the Product Page loads without reviews rather than crashing or freezing (graceful degradation).
- **Retries:** Use exponential backoff with jitter for transient errors (e.g., network timeout) before failing the transaction.

6.5 Disaster Recovery (DR) Plan

- **Strategy:** Active-Passive (Warm Standby) Multi-region deployment. The Passive region maintains running instances and up-to-date data replicas.
- **RPO (Recovery Point Objective):** > 15 minutes (Achieved via asynchronous database replication and frequent hourly backups).
- **RTO (Recovery Time Objective):** > 1 hour (Achieved via automated K8s failover and DNS switchover scripts, minimizing downtime).

---

7. API Specification

7.1 Protocol Selection

- **REST (Synchronous):** Used for **Admin and Transactional APIs** (/orders, /payments, /inventory). Standard CRUD operations.
- **GraphQL/gRPC (Synchronous): GraphQL** is used for the **Storefront API** to allow clients (Web/Mobile) to fetch complex, nested data (Product, Reviews, Related Items) in a single optimized query.
- **Events (Asynchronous):** Used for **Order Orchestration** (OrderCreated, InventoryReserved) via Kafka.

7.2 Endpoints (REST Example)

Path	Method	Purpose	Idempotency	Error Codes
/api/v1/orders	POST	Create a new order.	**Key Required**	400, 403, 409 (Conflict - if key exists)
/api/v1/orders/{id}	GET	Retrieve specific order details.	N/A	404, 403
/api/v1/payments	POST	Process payment for a transaction.	**Key Required**	402 (Payment Failed)

7.3 Versioning

- **Strategy: URI Versioning** (/api/v1/orders).
- **Rationale:** Provides clear, non-breaking contracts to external consumers and simplifies parallel deployment for new major feature releases.

---

8. Observability

Observability is built upon the three pillars to ensure rapid detection and diagnosis of issues in the distributed environment.

Pillar	Technology/Tool	Data Collected	Use Case
Logs	ELK Stack (Elasticsearch, Logstash, Kibana) / Datadog	Structured JSON logs from all services.	Root cause analysis, detailed debugging of error states.
Metrics	Prometheus & Grafana	**RED** (Rate, Errors, Duration) metrics for every service, CPU/Memory utilization, Queue depth.	Health monitoring, capacity planning, SLO tracking.
Traces	OpenTelemetry / Jaeger	Trace ID and Span context propagation across all service calls.	Identifying latency bottlenecks in distributed Sagas across multiple services.

8.1 SLOs/SLIs and Alerting Runbook

- **SLI (Service Level Indicators):**
  - **Availability:** Success rate of critical API calls (e.g., POST /orders success rate $> 99.9\%$).
  - **Latency:** P95/P99 latency of API calls (e.g., P99 of /search must be ≤ 200ms).
- **SLO (Service Level Objective):** Critical service availability target set at **99.9%**.
- **Alerting Runbook:** Automated alerts (e.g., PagerDuty) triggered when SLI breaches SLO. Each alert contains a link to a runbook detailing immediate triage steps (e.g., check database replication lag, scale up Cart Service replicas, analyze Kafka consumer lag).

---

9. Tech Stack Justification

Tier	Selected Option(s)	Alternatives Considered	Trade-offs / Justification
Backend	**Go, Java/Spring Boot**	Node.js, Python	**Go** for high-throughput, low-latency services (Checkout, Cart). **Java/Spring** for complex, long-running business logic (Order Orchestration, Admin) due to its maturity and strong enterprise features.
Frontend	**React**	Vue.js, Angular	Rich ecosystem, strong component model, largest pool of experienced developers for rapid development.
Data/Search	**Postgres, Mongo, Elasticsearch, Redis**	MySQL, Couchbase, Cassandra	**Polyglot Persistence:** Optimal performance/consistency profile achieved by matching the right tool to the domain need.
Messaging (Event Fabric)	**Kafka**	RabbitMQ, ActiveMQ	High throughput, durability, persistent event log required for event choreography and **replayability**

			(Sagas), making it the central nervous system.
Infrastructure	**Kubernetes (K8s)**	Docker Swarm, ECS	Industry standard for microservices orchestration, provides advanced networking, auto-scaling, and operational resilience features, reducing vendor lock-in.