2026 Roadmap Export

This document is an export of:

- docs/features/todo/2026/README.md
- docs/features/todo/2026/01-unified-order-state-engine-v1.md
- docs/features/todo/2026/02-menu-as-versioned-artifact-v1.md
- docs/features/todo/2026/03-kitchen-display-tv-v1.md
- docs/features/todo/2026/04-voice-ordering-v1-constrained.md
- docs/features/todo/2026/05-payments-moments-v1.md
- docs/features/todo/2026/06-partner-integrations-event-driven.md
- docs/features/todo/2026/07-ai-menu-insights-v1-readonly.md
- docs/features/todo/2026/08-menu-experiments-ab-testing.md
- docs/features/todo/2026/09-auto-pay-and-leave-v1.md

2026 Feature Roadmap (TODO)

This folder breaks down the 2026 roadmap into distinct, implementable features with:

- GIVEN / WHEN / THEN acceptance criteria
- Checkbox task lists for progress tracking
- Cross-references to existing feature docs in `docs/features/`

Current System State (baseline)

- **Order lifecycle exists (state machine)**
  - `Ordr` uses AASM with statuses: `opened → ordered → preparing → ready → delivered → billrequested → paid → closed`.
  - Order mutation history exists as `Ordraction` records (actions like `additem`, `removeitem`, `requestbill`).
- **Real-time infra exists and is in production use**
  - ActionCable channels exist: `OrdrChannel` and `KitchenChannel`.
  - Kitchen dashboard (TV-oriented) is implemented.
  - See:
    - `../done/REALTIME_IMPLEMENTATION_STATUS.md`
    - `../done/KITCHEN_DASHBOARD_UI.md`
- **Voice ordering exists (customer pages), but is not "constrained-intent engine" yet**
  - Smartmenu customer voice commands are documented and partially implemented.
  - See:
    - `../in-progress/voice-menus.md`
    - `../todo/voiceApp.md`
- **Payments exist (customer pay), but not "payments moments" / stored credentials**
  - Stripe PaymentIntent creation exists (`Payments::IntentsController`).
  - SaaS billing (restaurant subscription) is separately scoped.
  - See:
    - `../todo/stripe_restaurant_payments/README.md`
    - `../todo/bill-splitting-feature-request.md`
    - `../todo/auto-pay-and-leave.md`
- **Menus are "data models", not yet "versioned artifacts"**
  - Sorting, time restrictions, localization, OCR import exist.
  - No immutable menu versions or diffs.
  - See:
    - `../done/MENU_SORTING_IMPLEMENTATION.md`
    - `../done/MENU_TIME_RESTRICTIONS.md`

2026 Feature Specs Index

- `01-unified-order-state-engine-v1.md`

- 02-menu-as-versioned-artifact-v1.md
- 03-kitchen-display-tv-v1.md
- 04-voice-ordering-v1-constrained.md
- 05-payments-moments-v1.md
- 06-partner-integrations-event-driven.md
- 07-ai-menu-insights-v1-readonly.md
- 08-menu-experiments-ab-testing.md
- 09-auto-pay-and-leave-v1.md

Unified Order State Engine (v1)

Purpose

Make **one live, authoritative dining state** the foundation for:

- Customer ordering UX
- Staff + kitchen operational truth
- Voice actions (customer + staff)
- Payment state transitions
- Partner integrations (event-driven)

This spec introduces an **append-only canonical event log** and a **deterministic reducer** that produces a materialized "current state".

Current State (today)

- The system already has **order lifecycle state** in `Ordr` (AASM + enum): `opened → ordered → preparing → ready → delivered → billrequested → paid → closed`.
- The system records some user actions via `Ordraction` (e.g., `additem`, `removeitem`, `requestbill`).
- The system already has real-time broadcasting for orders and kitchen updates.

Gaps:

- `Ordraction` is not a **general canonical event log** (limited action types, not strictly append-only for all state mutations).
- There is no deterministic reducer / projection layer that can rebuild state from events.
- Not all state mutations are guaranteed to emit a single canonical record.

Cross references:

- `docs/features/done/REALTIME_IMPLEMENTATION_STATUS.md`
- `docs/features/done/KITCHEN_DASHBOARD_UI.md`

Scope (v1)

- Introduce `OrderEvent` as canonical event stream for dining state.
- Ensure **every order mutation** emits exactly one canonical `OrderEvent`.
- Provide deterministic reducer logic that can rebuild state.
- Provide at least one projection path:
  - `Ordr` + `Ordritem` continue to exist as the "materialized state", but are now **derived** and kept consistent via event-driven projection.

Non-goals (v1)

- Complex conflict-free replicated data types (CRDTs).
- Partner integrations.
- AI-driven automation.
- Multi-order/table merge/split.

Data Model (proposed)

`OrderEvent`

Fields (minimum viable):

- `id`
- `ordr_id` (required)
- `event_type` (e.g. `item_added`, `item_removed`, `status_changed`, `bill_requested`, `paid`)
- `entity_type` (`order`, `item`, `participant`, `payment`)
- `entity_id` (nullable, depending on entity)
- `payload` (JSON: qty, menuitem_id, modifiers, delay_minutes, etc.)
- `source` (`guest`, `staff`, `voice`, `system`, `webhook`)
- `idempotency_key` (nullable but recommended)
- `created_at`

Acceptance Criteria (GIVEN / WHEN / THEN)

Canonical event creation

- GIVEN an existing `Ordr` WHEN a customer adds an item through the UI THEN a single `OrderEvent` is created with `event_type=item_added`, `source=guest`, and payload including `menuitem_id` and `qty`.
- GIVEN an existing `Ordr` WHEN staff transitions an order from `ordered` to `preparing` THEN a single `OrderEvent` is created with `event_type=status_changed`, `source=staff`, and payload including `from=ordered` and `to=preparing`.
- GIVEN an existing `Ordr` WHEN a payment intent succeeds (via webhook) THEN a single `OrderEvent` is created with `event_type=paid`, `source=webhook`, and payload including `provider=stripe` and the external reference.

Deterministic reduction

- GIVEN a sequence of `OrderEvent` records for an `Ordr` WHEN the reducer replays events in ascending `created_at,id` order THEN the computed state is deterministic and identical for every replay.

- GIVEN an `OrderEvent` stream WHEN the reducer encounters an unknown `event_type` THEN the reducer does not corrupt state and the event is reported as unsupported.

Projection guarantees

- GIVEN that `OrderEvent` is enabled WHEN any controller/service mutates an order or its items THEN the mutation path emits an `OrderEvent` before returning success.
- GIVEN an emitted `OrderEvent` WHEN projections are processed THEN the materialized `Ordr` / `Ordritem` state reflects the event.

Progress Checklist

- ☐ Add `order_events` table + model (`OrderEvent`)
- ☐ Define event type taxonomy (v1 whitelist)
- ☐ Add idempotency strategy (recommended for voice + webhooks)
- ☐ Implement reducer (`OrderStateReducer`) with deterministic ordering
- ☐ Implement projection worker(s) (Sidekiq): apply events to `Ordr` / `Ordritem`
- ☐ Update existing mutation code paths (controllers/services) to emit `OrderEvent`
- ☐ Ensure ActionCable broadcasts are driven by projected state changes
- ☐ Add tests:
  - ☐ event creation for add/remove/status transitions
  - ☐ reducer determinism
  - ☐ projection correctness
- ☐ Add basic observability:
  - ☐ log event emission failures
  - ☐ metrics for projection lag

Menu as Versioned Artifact (v1)

Purpose

Make the menu **auditable, immutable-by-version, and explicitly activatable**.

This creates a moat: menus become "software" with:

- Immutable versions
- Diffs between versions
- Activation windows (time-based; A/B optional later)

Current State (today)

- Menus/sections/items are persisted as mutable records (`Menu`, `Menusection`, `Menuitem`).
- Sorting and time-window restrictions exist.
- Localization exists (menu/section/item locale tables).
- OCR import exists.

Gaps:

- No immutable `MenuVersion` concept.
- No diffs, no audit history of *what changed and when*.
- No activation windows.

Cross references:

- `docs/features/done/MENU_SORTING_IMPLEMENTATION.md`
- `docs/features/done/MENU_TIME_RESTRICTIONS.md`

Scope (v1)

- Introduce `MenuVersion` (immutable snapshot).
- Allow creating a new version from current menu state.
- Allow activating a specific version, optionally with a time window.
- Provide diff capability between any two versions.

Non-goals (v1)

- AI insights or auto-optimization.
- Automatic rollout suggestions.
- Multi-restaurant shared version graphs.

Conceptual Model

- `Menu` remains the logical container.
- `MenuVersion` is a frozen snapshot of:
  - menu metadata

  - sections
  - items
  - relevant fields (including prep time, allergens metadata references, margin hints)

Approach options:

- **Snapshot JSON** (fastest to ship): store a canonical JSON document per version.
- **Versioned tables** (more relational): `menu_versions`, `menu_section_versions`, `menu_item_versions`.

Acceptance Criteria (GIVEN / WHEN / THEN)

Version creation

- GIVEN a menu with sections and items WHEN a user creates a new menu version THEN a `MenuVersion` is created containing an immutable snapshot of all sections/items and menu metadata.
- GIVEN an existing `MenuVersion` WHEN a user attempts to edit it THEN the system prevents mutation (read-only).

Diffing

- GIVEN two menu versions A and B WHEN the diff is requested THEN the system returns a deterministic diff describing:
  - added/removed sections
  - added/removed items
  - changed fields (name/price/description/availability metadata)

Activation windows

- GIVEN a menu version with an activation window WHEN the current time is within the window THEN the smart menu displays that version.
- GIVEN a menu version with an activation window WHEN the current time is outside the window THEN the smart menu displays the default active version.

Progress Checklist

☐ Decide storage strategy: JSON snapshot vs versioned tables
☐ Add `menu_versions` model/table
☐ Implement "Create Version" action from current menu
☐ Implement "Activate Version" (manual)
☐ Implement activation window fields (`starts_at`, `ends_at`)
☐ Implement version selection logic in smart menu rendering
☐ Implement `MenuVersionDiffService`

- ☐ Add admin/staff UI for:
  - ☐ list versions
  - ☐ diff view
  - ☐ activate/schedule activation
- ☐ Add tests for snapshot integrity + activation logic

Kitchen Display (TV-Optimised) (v1)

Purpose

Make the kitchen display a **first-class operational truth surface**:

- Large-format, glanceable, TV-optimized
- Real-time updates only (no polling)
- Supports "voice-ready visual state" (IDs visible, stable identifiers)

Current State (today)

- A TV-optimized kitchen dashboard is already implemented.
- Uses ActionCable + `KitchenChannel` + broadcasting.
- Shows orders by status in three columns and supports one-click status changes.

Cross references:

- `docs/features/done/KITCHEN_DASHBOARD_UI.md`
- `docs/features/done/REALTIME_IMPLEMENTATION_STATUS.md`

Gap vs 2026 roadmap

- Current dashboard is strongly aligned with the roadmap goals.
- What's missing for the roadmap framing is mostly:
  - formalization as "the operational truth" surface
  - stronger stable identifiers for voice workflows (table codes, order IDs, participant IDs)
  - alignment with the future event-driven order state engine (when introduced)

Scope (v1)

- Keep existing dashboard.
- Add explicit identity and UX affordances for voice + staff workflows:
  - consistent display of: `order_id`, `table_id`, optional `short_code`
  - clear states for "bill requested", "paid", "closed" visibility rules
- Ensure it is driven by the same canonical state as guest/staff UIs.

Non-goals (v1)

- Kitchen staffing, assignments, or messaging.
- Expo-style bump screens.
- Inventory integration.

Acceptance Criteria (GIVEN / WHEN / THEN)

- GIVEN the kitchen dashboard is open on a TV WHEN a new order is placed by a customer THEN the order appears without refresh and is visible within 1 second of commit.
- GIVEN the kitchen dashboard is open WHEN an order changes from `ordered` to `preparing` THEN the order card moves to the correct column without refresh.
- GIVEN an order card is visible WHEN staff need to reference it for voice or troubleshooting THEN the card displays stable identifiers:
  - `order_id`
  - `table_id` (and/or table label)
  - at least one additional stable reference (e.g., `created_at` or a human short code)
- GIVEN the customer smart menu is open for the same table WHEN the kitchen dashboard shows order status `ready` THEN the customer view reflects `ready` state consistently (same truth).

Progress Checklist

- ☐ Audit current kitchen dashboard behavior vs roadmap requirements
- ☐ Ensure KDS shows stable identifiers clearly (order/table/participant references)
- ☐ Confirm "no polling" guarantee remains true
- ☐ Define explicit rules for which terminal states remove cards (delivered/paid/closed)
- ☐ Align broadcasting payload with unified order state model (future)
- ☐ Add smoke tests for ActionCable updates to KDS

Voice Ordering (v1 – Constrained)

Purpose

Ship a **hard-whitelist** voice intent system that:

- Saves real seconds in service
- Produces predictable outcomes
- Is always reversible within a short window
- Always emits canonical order state mutations

Current State (today)

- Smartmenu customer voice commands exist and are documented.
- The implementation supports voice capture, async processing, intent parsing, and action execution in the Smartmenu context.

Cross references:

- `docs/features/in-progress/voice-menus.md`
- `docs/features/todo/voiceApp.md`

Gaps vs this spec:

- Intent set is broader / less formally constrained than the roadmap.
- There is no explicit system-wide "confidence < threshold => reject" rule enforced everywhere.
- There is no unified "undo window" across voice actions.
- Voice actions are not yet formally bound to a canonical OrderEvent stream (future: `OrderEvent`).

Scope (v1)

Supported intents (hard whitelist)

- `add_same_item`
- `undo_last_action`
- `request_bill`
- `pay_now`
- `order_ready` (staff)
- `delay_order_x` (staff)

Execution rules

- Confidence `< 0.85` => reject with a clear UI response (no mutation).
- Always emit a canonical state mutation (future: `OrderEvent`, currently: at least consistent `Ordraction` + state transition).
- Always show confirmation.
- Undo window: 5–10 seconds.

Non-goals (v1)

- No free-text ordering.
- No menu discovery by voice.
- No "ask anything" conversational layer.

Acceptance Criteria (GIVEN / WHEN / THEN)

Confidence gating

- GIVEN a voice transcript classified as one of the whitelisted intents WHEN the confidence is `< 0.85` THEN the system does not mutate the order and returns a rejection response with a visual fallback.

add_same_item

- GIVEN a customer has an active order and has previously added an item WHEN the customer says "same again" and the system resolves intent `add_same_item` with confidence `>= 0.85` THEN the system adds `qty=1` of the most recently added item and returns a confirmation.
- GIVEN `add_same_item` is executed WHEN the mutation succeeds THEN a canonical mutation record is written (future: `OrderEvent.item_added`).

undo_last_action

- GIVEN a voice action mutated the order within the last 10 seconds WHEN the customer says "undo" and intent is `undo_last_action` THEN the system reverts the last mutation and returns confirmation.

request_bill

- GIVEN an order has no `opened` items and has at least one ordered item WHEN a customer says "can we get the bill" and intent is `request_bill` THEN the system transitions the order to `billrequested` and returns confirmation.

pay_now

- GIVEN an order is in `billrequested` WHEN the customer says "pay now" and intent is `pay_now` THEN the system presents the payment UX entry point and records the intent.

staff intents

- GIVEN a staff user is authenticated and scoped to the restaurant WHEN staff says "order ready" with a table/order identifier THEN the order transitions to `ready` and the kitchen/customer views update in real time.

- GIVEN staff says "delay order 10 minutes" WHEN intent is `delay_order_x` THEN a delay marker is recorded and visible in operational UI.

Progress Checklist

- ☐ Define canonical intent whitelist (single source)
- ☐ Implement confidence gating consistently
- ☐ Implement undo window + rollback strategy
- ☐ Add staff-authenticated voice endpoints (separate from customer)
- ☐ Ensure each voice intent emits canonical order mutation records
- ☐ Add integration tests for each intent + rejection path
- ☐ Document "Do Not Build" rules in this feature scope

Payments Moments (Stripe-led) (v1)

Purpose

Make payment a **moment, not a hunt**.

Key principle:

- The system owns **when** payment happens (state + UX triggers), not necessarily **how** (Stripe implementation details can evolve).

Current State (today)

- Stripe is used for end-customer payments via `Payments::IntentsController` (PaymentIntent creation).
- Order lifecycle includes `billrequested`, `paid`, `closed` states.
- There is existing work scoped for auto-pay and bill splitting.

Cross references:

- `docs/features/todo/stripe_restaurant_payments/README.md` (note: this is SaaS billing; separate)
- `docs/features/todo/bill-splitting-feature-request.md`
- `docs/features/todo/auto-pay-and-leave.md`

Gaps:

- No explicit "request bill" UX moment definition across surfaces.
- Partial payments are not implemented.
- "Pay at table" QR-to-checkout flow may exist partially, but not formalized as an end-to-end product moment.

Scope (v1)

- Standardize the **bill requested** transition as the trigger to show payment UI.
- Implement pay-at-table flow:
  - QR → Stripe Checkout or Payment Element
- Implement partial payment support (v1): **even split only**.

Non-goals (v1)

- Itemized splitting.
- Stored credentials / auto-pay (separate feature: Auto-Pay-and-Leave).
- POS integration.

Acceptance Criteria (GIVEN / WHEN / THEN)

Request bill moment

- GIVEN an order with at least one ordered item and no opened items WHEN the customer triggers "Request Bill" THEN the order transitions to `billrequested` and the UI exposes payment options.
- GIVEN an order is in `billrequested` WHEN the customer reloads the smart menu THEN `payVisible=true` (or equivalent) is true and payment entry remains visible.

Pay at table (Stripe)

- GIVEN an order is in `billrequested` WHEN the customer chooses "Pay now" THEN the system creates a Stripe PaymentIntent (or Checkout Session) and returns a client secret/session URL.
- GIVEN a Stripe payment succeeds WHEN confirmation is received THEN the order transitions to `paid` and the kitchen/staff/customer views update in real time.

Even split (v1)

- GIVEN an order in `billrequested` with N participants WHEN the customer selects "Split evenly" THEN the system computes N equal amounts (with rounding rules) and generates a payment flow per participant.
- GIVEN some participant payments succeed and others are pending WHEN staff views the order THEN staff can see partial settlement status (e.g., `paid_portion`, `remaining`).

Progress Checklist

- ☐ Define canonical payment UX states tied to order status (`billrequested` as entry)
- ☐ Implement pay-at-table UX entry point and server endpoints
- ☐ Add Stripe integration for chosen flow (Payment Element / Checkout)
- ☐ Add payment confirmation handling (success/failure)
- ☐ Add even-split data model and UI (reuse/align with bill splitting doc)
- ☐ Add staff visibility of split payment state
- ☐ Add tests:
  - ☐ request bill transition => pay visible
  - ☐ successful payment => order becomes paid
  - ☐ even split calculations and rounding

Partner Integrations (Event-driven, Replaceable) (v1)

Purpose

Expose mellow's "live dining state" as a set of **event-driven, loosely coupled, replaceable** integrations.

Focus:

- You emit reliable signals.
- Partners consume signals to make decisions.
- No partner UI duplication inside mellow.

Current State (today)

- Real-time events exist internally (ActionCable) for app UIs.
- Stripe customer payment intents exist (creation), but webhook-to-order-event mapping is not formalized.
- No first-class partner event stream exists.

Cross references:

- `docs/features/done/REALTIME_IMPLEMENTATION_STATUS.md`
- `docs/features/todo/stripe_restaurant_payments/README.md` (SaaS billing – separate)

Scope (v1)

Integration types

- **Stripe deepened (payments)**
  - Payment webhooks → canonical order events
- **Workforce (Nory-class)**
  - Provide: order velocity, item prep times, table occupancy duration
- **Reservations/CRM (SevenRooms-class)**
  - Provide: in-meal behaviour, order pacing, time-to-pay
- **Messaging (Twilio-class)** (later quarter, but specified here as a partner type)
  - Provide: receipts + nudges

Architectural requirements

- All integrations are:
  - Event-driven
  - Loosely coupled (behind adapters)
  - Replaceable

Non-goals (v1)

- No UI for partners.
- No "two-way sync" beyond identity context (initially).

Acceptance Criteria (GIVEN / WHEN / THEN)

Canonical partner event stream

- GIVEN an order mutation occurs WHEN the system emits canonical events THEN an integration adapter can subscribe and receive a normalized payload.
- GIVEN the Stripe webhook receives `payment_intent.succeeded` WHEN the webhook is processed THEN a canonical order payment event is emitted and downstream adapters are invoked.

Workforce signals

- GIVEN a restaurant has active orders WHEN workforce integration polling/export is requested THEN the system can provide:
  - orders per minute (velocity)
  - prep time estimates by item
  - table occupancy duration

CRM signals

- GIVEN an order transitions to `billrequested` WHEN CRM adapter receives the event THEN it can infer "time-to-pay" tracking start.

Progress Checklist

- ☐ Define canonical event payload schema for partners
- ☐ Implement integration adapter interface (`PartnerIntegrationAdapter`)
- ☐ Implement Stripe webhook mapping → canonical events
- ☐ Implement Workforce export endpoint (read-only)
- ☐ Implement CRM export endpoint (read-only)
- ☐ Add configuration per restaurant for enabling integrations
- ☐ Add observability:
  - ☐ per-adapter success/fail metrics
  - ☐ dead-letter logging

AI Menu Insights (v1 – Read-only)

Purpose

Provide operational insight that restaurants don't have today, without auto-changing the menu.

This is a read-only "insights layer" derived from live dining signals.

Current State (today)

- Analytics and reporting capabilities exist in the system (including materialized view patterns in other parts of the codebase).
- Voice commands are tracked (VoiceCommand persistence exists per voice docs).
- Order lifecycle and kitchen operational events exist.

Cross references:

- `docs/features/in-progress/voice-menus.md` (voice capture + outcomes)
- `docs/features/done/ordering.md` (analytics dashboard plan)

Gaps:

- No unified event stream to reliably feed insights.
- No dedicated "Insights" UI and no defined metrics set for menu-level insight.

Scope (v1)

Insights to deliver (read-only):

- Slow movers (items that rarely get ordered)
- Prep-time bottlenecks (items that correlate with long prep times)
- Voice-trigger frequency (which items users attempt via voice)
- Abandonment points (where people stop short of ordering or paying)

Non-goals (v1)

- No auto menu changes.
- No AI-driven A/B creation.
- No prescriptive optimization.

Acceptance Criteria (GIVEN / WHEN / THEN)

Slow movers

- GIVEN a menu with historical orders WHEN an owner views "Slow movers" THEN the system shows items sorted by lowest order frequency

over a selectable window.

Prep-time bottlenecks

- GIVEN order item timestamps are available (ordered → preparing → ready) WHEN an owner views "Prep-time bottlenecks" THEN the system ranks items by median time-to-ready and highlights outliers.

Voice-trigger frequency

- GIVEN voice ordering is enabled for a restaurant WHEN an owner views voice insights THEN the system shows the top N items referenced by voice commands, including success/failure rate.

Abandonment points

- GIVEN customers can browse and request bills/pay WHEN an owner views abandonment insights THEN the system shows drop-off at key steps:
  - menu viewed → item added
  - item added → order submitted
  - bill requested → payment started
  - payment started → payment succeeded

Progress Checklist

☐ Define metrics contract and time windows
☐ Define data sources (orders, order items, voice commands)
☐ Add aggregation layer (materialized views or cached queries)
☐ Add UI surface (owner dashboard tab)
☐ Add export (CSV/JSON)
☐ Add tests for aggregations and edge cases (no data)

Menu Experiments (A/B + Time-boxed) (v1)

Purpose

Enable safe experimentation on menus:

- A/B testing between menu versions
- Time-boxed experiments
- Explicit audit trail

This builds directly on **Menu as Versioned Artifact**.

Current State (today)

- There is no `MenuVersion` system yet.
- Menu time restrictions exist, but that is not a version experiment system.

Cross references:

- `docs/features/todo/2026/02-menu-as-versioned-artifact-v1.md`
- `docs/features/done/MENU_TIME_RESTRICTIONS.md`

Scope (v1)

- Experiments are defined as:
  - one control version
  - one variant version
  - allocation (e.g., 50/50)
  - eligibility rules (time window, optionally table-based)
  - start/end timestamps
- The system selects a menu version at request time based on experiment assignment.
- The system records exposure events for analysis.

Non-goals (v1)

- AI suggestions.
- Auto-optimization.
- Multi-variant experiments.

Acceptance Criteria (GIVEN / WHEN / THEN)

Experiment creation

- GIVEN a menu has at least two immutable versions WHEN an owner creates an experiment with control=V1 and variant=V2 THEN the system persists an experiment record with allocation rules and a time window.

Version assignment

- GIVEN an experiment is active and a customer opens the smart menu WHEN version assignment is computed THEN the customer is deterministically assigned to either control or variant based on a stable key (e.g., session id).
- GIVEN a customer refreshes the page WHEN version assignment is recomputed THEN the same customer receives the same version during the experiment window.

Exposure logging

- GIVEN a customer receives the variant version WHEN the menu is rendered THEN an exposure event is recorded with: menu_id, version_id, experiment_id, session_id, timestamp.

Safety

- GIVEN the experiment window ends WHEN a customer opens the smart menu THEN the system serves the default active version and no longer assigns variants.

Progress Checklist

- ☐ Implement MenuVersion system (dependency)
- ☐ Add `menu_experiments` table/model
- ☐ Define deterministic assignment strategy (session-based)
- ☐ Add exposure logging (table or event stream)
- ☐ Update smart menu rendering to select version
- ☐ Add reporting for experiment results (basic)
- ☐ Add tests:
  - ☐ deterministic assignment
  - ☐ window enforcement
  - ☐ exposure logging

Auto Pay & Leave (v1)

Purpose

Close the dining loop so the customer can **pay and leave without waiting**, while keeping staff in control.

This feature is opt-in and must be explicit, safe, and observable.

Current State (today)

- Order status includes `billrequested`, `paid`, `closed`.
- Stripe PaymentIntent creation exists.
- There is already an existing TODO feature doc for auto-pay-and-leave.

Cross references:

- `docs/features/todo/auto-pay-and-leave.md`
- `docs/features/todo/bill-splitting-feature-request.md`
- `docs/features/todo/stripe_restaurant_payments/README.md` (SaaS billing – separate)

Gaps:

- Stored credentials per diner are not implemented.
- Webhook-driven payment intent -> order state transition is not formalized.
- No "table freed automatically" mechanics.

Scope (v1)

- Customer can opt-in to store a payment method (Stripe-managed) during the meal.
- Customer can opt-in to auto-pay.
- Trigger auto-pay when:
  - meal is marked complete, or
  - bill is requested (depending on rules)
- On success:
  - mark order paid
  - send receipt
  - mark table freed (if/when table state exists)

Non-goals (v1)

- Forced auto-pay.
- Itemized bill splitting.
- Offline support.

Acceptance Criteria (GIVEN / WHEN / THEN)

Opt-in capture

- GIVEN a customer is viewing an active order WHEN they choose "Add payment method" THEN a Stripe-managed UI captures the payment method and the system stores only a provider reference (no PAN/PII).

Arming auto-pay

- GIVEN a payment method is on file WHEN the customer enables auto-pay THEN the system records consent and shows a clear confirmation.

Auto capture

- GIVEN auto-pay is enabled and an order becomes chargeable WHEN the trigger occurs (bill requested OR meal complete) THEN the system attempts a Stripe capture and records success/failure.
- GIVEN auto-pay capture succeeds WHEN confirmation is received THEN the order becomes `paid` and staff UI receives a real-time notification.
- GIVEN auto-pay capture fails WHEN failure is received THEN the order remains open and staff UI receives a real-time notification with a non-sensitive failure reason.

Table freed

- GIVEN an order is successfully paid WHEN "auto pay & leave" is enabled THEN the table is marked available/freed according to the restaurant's table management rules.

Progress Checklist

- ☐ Reconcile this spec with `docs/features/todo/auto-pay-and-leave.md` (merge/keep both with links)
- ☐ Implement payment method on file (Stripe customer/payment method)
- ☐ Add consent + arming flags to order/participant
- ☐ Implement AutoPayCaptureJob
- ☐ Implement webhook mapping to update order state
- ☐ Add staff notifications (ActionCable)
- ☐ Add customer receipt delivery mechanism (email/SMS) (see Messaging feature)
- ☐ Add tests for:
  - ☐ consent persistence
  - ☐ capture success -> order paid
  - ☐ capture failure -> staff notified