Business Portfolio Case Study: Onsite Scheduling Architecture

Designing a validated scheduling decision layer for onsite service work

ROLE

Process analysis & prototype architecture (solo)

STACK

Python · Pydantic · SQLite · Flask · FastAPI · pytest · Sentry

STATUS

Sandbox prototype · synthetic data · 115 passing tests

Abstract

A custom manufacturing company wanted to reduce friction in hourly on-site service booking by removing manual judgment from scheduling. The initial goal was to make booking feel like selecting an available appointment slot, but the diagnostic finding was that calendar automation was premature: time allocation was unreliable because estimates depended on intuition, inconsistent inputs, and limited usable operating data.

The intervention I proposed was to formalize the field-assessment intake, introduce the data capture needed to support future statistics and KPIs, and build a prototype scheduler to test how scheduling logic could be derived from that structure. The documentation, recommendations, and prototype build were handed to the company. This portfolio case study rebuilds the same principle from memory, using no real customer data or company-specific internal documentation.

1. Business Context

The company offered hourly on-site service as one of its order-fulfilment options, delivered by two in-house teams. The workflow relied on a legacy ERP/order system, printed job forms, binders, email updates, and recurring verbal coordination between customer service, the service manager, production, and field teams.

Before each service job, a pre-service field assessment captured photos, measurements, marked-up images, and practical job details. These materials were used to prepare the job forms, estimate the service time slot, and the approximate service charge to be included in the customer quote before order approval.

In practice, the assessment outputs were not converted into structured scheduling rules. Time allocation remained judgment-based, and stakeholders reported frequent misallocation, often in the customer's favour. When estimates were wrong, the issue created operational friction across quoting, scheduling, production readiness, field-team execution, completion reporting, and invoicing.

The workflow showed that the problem was not simply "finding an open calendar slot." The company first needed a more reliable way to capture field-assessment data, define readiness conditions, track estimate accuracy, and translate job characteristics into duration and team-placement logic.

2. Diagnostic Insight

The core insight was that service duration should not be treated as a free-form guess. It should be derived from structured job-specific inputs, then calibrated against actual time worked.

The first requirement was a simple, coherent pre-service field assessment intake with minimal free-text entry and mostly controlled fields, numeric inputs, and standardized categories. This would make the field assessment easier to translate into a usable operating dataset.

Useful starting inputs included surface type or surface complexity, service area in square feet, material complexity, required team type, and practical job constraints. Due date was also important, but as a scheduling constraint rather than a duration input.

These inputs would not create a perfect estimate on their own. Their value was that they created a transparent and calibratable structure. Instead of relying on individual judgment hidden in notes, emails, or memory, the business could move toward a repeatable scheduling method:

> capture the right inputs → validate completeness → estimate duration → schedule only if ready → record the decision → compare estimate against actual time worked.

That measurement loop was the foundation required before calendar automation or AI assistance could be trusted. Without baseline statistics and KPIs, the business could not know whether scheduling accuracy was improving or whether automation was simply reproducing unreliable judgment faster.

I would start with three metrics:

- mean absolute error in hours between estimated and actual time,
- percentage of jobs completed within ±1 hour of the estimate,
- mean signed error to show whether the company systematically under- or over-estimated service duration.

3. Prototype Approach

The prototype was designed to test the scheduling decision layer, not to replace the company's full operating system. The goal was to determine whether service booking could be made more reliable

by translating structured field-assessment inputs into duration estimates, team placement, readiness checks, and auditable scheduling outcomes before connecting to any external calendar or AI layer.

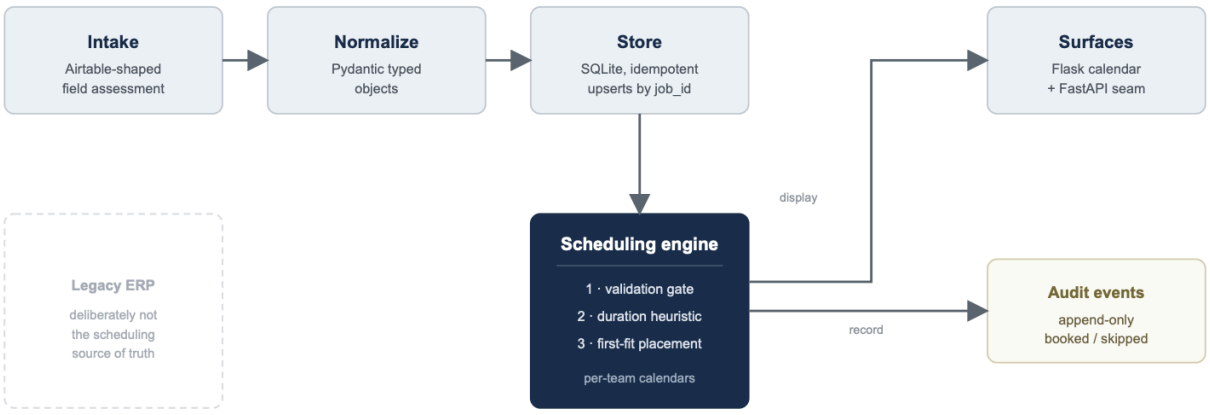

Figure 1 — System architecture. A clean intake surface feeds a normalization boundary and a single store; the deterministic engine reads from the store, writes an append-only audit event for every decision, and exposes results through a local UI and an API seam. The legacy ERP is intentionally outside the decision path.

The first architectural decision was to separate scheduling intake from the legacy ERP. Instead of asking the scheduler to parse scattered operational notes, the prototype used an Airtable-shaped intake structure with only the fields needed for scheduling: job details, field-assessment outputs, team assignment, due date, and on-site service status. This created a clean input surface without claiming to replace the ERP.

Raw intake data then passed through a normalization boundary before entering the scheduling logic. This step converted inconsistent statuses, blanks, linked-record formats, duration units, and known input errors into typed, predictable objects. The business reason was simple: the scheduling engine should not have to make decisions on messy records.

A local SQLite store acted as the prototype's single source of truth. This kept the sandbox lightweight while still making jobs, assessments, scheduled on-site services, and audit events queryable in one durable place. Repeated syncs used idempotent upserts, so re-running the prototype would update existing records rather than create duplicates.

The core scheduling logic used a deterministic duration heuristic. Service duration was calculated from structured assessment inputs such as surface complexity, service area, material complexity, and team-specific offsets. This was not presented as a calibrated production model. Its purpose was to prove that the estimate could be made explicit, explainable, and tunable instead of hidden inside individual judgment.

Before estimating or scheduling, the prototype applied a validation gate. A job could only move forward if required fields were present:

- pending status,
- assigned team,
- due date,
- completed assessment,
- the required duration inputs.

Incomplete jobs were skipped rather than forced into the calendar. This was a deliberate design choice: a missing input should stop the process instead of producing a false sense of confidence.

For placement, the scheduler used a first-fit greedy heuristic. It scanned each team's calendar independently, within business hours and before the job due date, then placed the job in the first available slot large enough for the estimated duration. Separate calendars prevented cross-team collisions and reflected the business reality that two teams could work in parallel.

Every booking or skip produced an append-only audit event. This made the decision reconstructable: managers could see whether a job was booked, skipped, or left pending, and why. The audit trail was intentionally separate from the input surface so decisions could be reviewed without silently overwriting the original intake.

ACTOR	EVENT	JOB	STATUS	DETAIL
scheduler	SKIPPED	recJOB3	PENDING → PENDING	Field assessment not completed
scheduler	BOOKED	recJOB2	PENDING → BOOKED	crew-b · 09:00–16:45 (7.75h)
scheduler	BOOKED	recJOB1	PENDING → BOOKED	crew-a · 09:00–15:30 (6.5h)

Figure 2 — Actual output of the engine on the synthetic sample jobs. Two jobs are booked into separate team calendars; one is skipped with its reason recorded rather than guessed.

The prototype included a local Flask calendar for demonstration and a FastAPI surface for later integration. The UI made the schedule searchable and visible by team, while the API created a seam where future automation or an assistant could trigger the same validated scheduling workflow.

A typical run followed a fixed sequence:

sync intake records → normalize data → validate readiness → estimate duration → place job in a team calendar → commit scheduled on-site service → write audit event → display schedule and logs

The demonstration scenario used synthetic sample jobs. Two jobs were successfully scheduled into separate team calendars, while one was skipped because the field assessment was incomplete. The skipped case was central to the prototype's logic: when required inputs were missing, the system declined to guess and recorded the reason.

This prototype demonstrates how a vague automation request can be translated into explicit business rules, validation logic, placement constraints, and auditable decisions. It also shows why calendar automation or AI assistance should sit on top of a trusted decision layer rather than replace the missing logic itself.

Technical evidence: the rebuilt prototype includes 115 passing pytest tests across normalization, SQLite persistence, duration calculation, size banding, validation, first-fit placement, team isolation, idempotent upserts, audit logging, and the Flask/FastAPI surfaces. This supports the reliability of the sandbox implementation, but does not claim production deployment, calibrated estimate accuracy, or measured business impact.

4. Production Readiness Considerations

Although this was a sandbox prototype, the architecture included several reliability decisions that would matter in a real scheduling environment. Scheduling systems are often run repeatedly through sync jobs, manual refreshes, or integration retries; if those runs create duplicate jobs, duplicate on-site service rows, or duplicate calendar events, operational trust breaks quickly.

The prototype handled local idempotency by upserting jobs, field assessments, and on-site service records by `job_id`, so repeated syncs updated existing records rather than creating duplicates. Team calendars were isolated during each scheduling run to prevent double-booking, and audit events were append-only so booking and skip decisions could be reviewed rather than silently overwritten.

A production version would need a stronger reliability layer around external integrations: stored external calendar event IDs, idempotency keys for scheduling runs, retry handling for partial API failures, sync-status tracking, reconciliation between the local schedule and external calendars, and alerts for jobs that repeatedly fail validation or cannot be placed before the due date.

Security and governance would also need to be expanded before deployment. Production access would require authentication, role-based permissions, secrets management, least-privilege access to calendar and source systems, data minimization, and clear handling of customer names, contact details, addresses, site photos, notes, logs, backups, and retention. Human overrides and integration writes would also need to be audited.

These considerations were outside the scope of the sandbox build, but they shaped the architecture: the prototype separates intake, validation, scheduling decisions, persistence, audit history, and integration seams so the system could be hardened before being connected to real operational data.

Conclusion

This case study demonstrates that the scheduling problem was not primarily a calendar or AI-agent problem, but a decision-architecture problem. The prototype shows how an intuition-based booking process can be reframed into structured intake, readiness validation, explainable duration logic, team-specific placement, and auditable outcomes. Its value is not in claiming production impact, but in showing the reasoning required before automation can be trusted: define the inputs, validate completeness, make the decision logic explicit, measure estimate accuracy, and only then connect the process to external calendars or AI assistance.

Confidentiality Note

This case study is fully anonymized and reconstructed from non-proprietary personal notes. It does not contain any private company files, screenshots, system exports, client records, or identifying operational details. Quantitative claims are presented as approximations where appropriate, and none should be read as measured or independently verified results.