**Goal**

Deliver a production-ready, multi-tenant booking platform for adventure/paintball parks with SSR Flask, Bootstrap, PostgreSQL, and GCP (Cloud Run + Cloud SQL), including payments and admin tooling.

**High-level phases**

* Discovery and requirements
* Architecture and data model
* Dev environment and repo setup
* Core backend (Flask, auth, bookings)
* Frontend (Jinja2 + Bootstrap)
* Payments (Stripe recommended)
* Admin tooling and calendar UX
* Ops (GCP infra, CI/CD, secrets, observability)
* QA, hardening, and launch

**Detailed implementation steps**

**1) Requirements and product scope**

* **Define user roles**: Customer, Park Admin, Platform Admin.
* **Core customer flows**: Browse park, pick date/time, select package/players, add-ons, pay, confirm, receive email/SMS.
* **Core admin flows**: Configure park/fields, schedule and availability, capacity rules, pricing, blackout dates, coupons, manual bookings, refunds, reports.
* **Multi-tenant model**: One platform, many parks; data isolation by park\_id; per-park branding and settings.
* **Payment flows**: Authorization, capture, refunds, cancellations, partial refunds; idempotency; webhook reliability.
* **Non-functional**: P95 page latency, cost targets, availability windows, GDPR/PII, audit logging, backups, error budgets.

**2) Architecture and repo layout**

* **Monorepo structure**:
* app/ (Flask app)
* app/templates/, app/static/ (Bootstrap, CSS, JS)
* app/blueprints/ (modular: public, booking, admin, webhooks)
* app/models/ (SQLAlchemy models)
* app/services/ (availability, pricing, emails, payments)
* app/tasks/ (async/background via Cloud Tasks or cron)
* migrations/ (Alembic)
* infra/ (Terraform optional or gcloud configs)
* tests/
* **Core components**:
* SSR Flask with Blueprints + Jinja2.
* SQLAlchemy + Alembic + psycopg2-binary.
* Stripe SDK + webhook blueprint.
* Bootstrap 5 assets compiled/minified (optional Sass).
* Email via SendGrid/Mailgun; SMS optional (Twilio).
* Cache layer: Flask-Caching with Redis (Cloud Memorystore optional later).
* **Security**: Session cookies (HTTPOnly, Secure), CSRF, rate limiting, RBAC.

**3) Data model (initial)**

* **Tenancy**: Every business entity references park\_id.
* **Entities**:
* Park: name, slug, timezone, brand settings.
* Field: belongs to Park, capacity, parallel sessions allowed.
* Package: name, duration, base price, min/max players.
* Addon: name, price, availability rules.
* TimeslotTemplate: day-of-week, open/close, slot duration, per-field capacity.
* Blackout: date range, reason.
* Booking: park, field, package, start/end, players, status, total, currency.
* BookingItem: addons per booking.
* Payment: provider, intent\_id, amount, currency, status, refunded\_amount.
* Coupon: code, discount type/value, constraints, usage count/limits.
* User: auth table; roles with UserRole referencing park\_id and role.
* AuditLog: actor, action, entity, before/after, timestamp.
* **Indexes**: (park\_id, start\_time), (park\_id, field\_id, start\_time), status, slug, foreign keys.
* **Timezones**: store UTC; display in park timezone.

**4) Dev environment and repo setup**

* **Initialize project**:
* Python 3.11+ venv, pip-tools or poetry.
* Base dependencies: Flask, SQLAlchemy, Alembic, Flask-WTF, Flask-Login, Flask-Caching, psycopg2-binary, python-dotenv, stripe, gunicorn, itsdangerous, requests.
* **Config management**:
* config.py with Development, Production classes.
* Read secrets from env; in prod, pull from Secret Manager.
* **Static assets**:
* Integrate designer HTML/CSS into templates/ and static/.
* Ensure Bootstrap 5 and basic components (navbar, cards, forms, modals).
* **Local DB**: Use Dockerized Postgres; alembic init, create migration scripts; seed data (demo park).

**5) Flask app foundation**

* **App factory**: create\_app(config\_name) pattern; register blueprints.
* **Auth**:
* Customer auth light (magic link or email/password).
* Admin auth with 2FA optional (TOTP) and RBAC by park\_id.
* CSRF via Flask-WTF; session lifetime; remember-me for admins off by default.
* **Error handling**: 404/500 pages; structured JSON for API routes.
* **Validation**: WTForms or pydantic for service-level validation.

**6) Booking engine (domain logic)**

* **Availability**:
* Generate slots from TimeslotTemplate + Blackout + per-field capacity.
* Concurrency-safe check when reserving: use transactional locks or uniqueness constraint on (park\_id, field\_id, start\_time, seat) or capacity counters with FOR UPDATE.
* **Pricing**:
* Base price from Package x players + addons - coupons + taxes/fees.
* Pluggable pricing strategy service.
* **Reservation lifecycle**:
* Statuses: initiated, pending\_payment, confirmed, cancelled, refunded.
* Hold inventory for a short TTL (e.g., 10 minutes) before payment; release on timeout (cron).
* **Payments** (Stripe recommended):
* Create PaymentIntent with amount/currency and capture on success.
* Handle SCA 3DS; return client secret to UI if needed.
* Webhooks for payment\_intent.succeeded, payment\_intent.payment\_failed, charge.refunded.
* Idempotency keys for retries; store provider IDs.
* **Refunds**:
* Admin-initiated partial/full refunds; reconcile Payment and Booking status.

**7) Customer-facing views (SSR)**

* **Pages**:
* Park landing: packages, fields, CTA.
* Availability calendar: pick date/time (month/week view).
* Booking details: players, addons, contact info, coupon.
* Checkout: review + payment; confirmation page.
* **UX**:
* Bootstrap forms; client-side validation; minimal JS.
* Timezone display and friendly formatting.
* Accessibility and mobile responsiveness.

**8) Admin console**

* **Dashboard**: KPIs (bookings, revenue, utilization).
* **Calendar**: drag-and-drop to move/cancel bookings, create manual holds; week/day views per field.
* **Configuration**: fields, packages, addons, timeslot templates, blackout dates, coupons, branding.
* **Bookings**: search, edit players, add addons, refund/cancel.
* **Reports**: CSV export; filters by date/field/package; per-park scopes.

**9) Emails and notifications**

* **Transactional emails**: Booking confirmation, reminder, cancellation, refund receipts.
* **Templates**: Jinja-based emails; store per-park branding.
* **Scheduling**: Reminders via Cloud Scheduler + Cloud Tasks or cron endpoint.

**10) Payments integration (deep)**

* **Stripe setup**:
* Products not required; compute price server-side.
* Webhook endpoint with signature verification; retry-safe; store events with dedupe.
* Reconciliation job to verify settlement vs bookings (daily).
* **Edge cases**:
* Payment succeeds but redirect lost: webhook confirms.
* Timeouts: release holds; notify customer.
* Chargeback flow: mark booking at risk; email admin.

**11) GCP infrastructure**

* **Services**:
* Cloud Run (Flask via Gunicorn).
* Cloud SQL for PostgreSQL (Private IP via Serverless VPC Access).
* Secret Manager for API keys (Stripe, email).
* Cloud Storage bucket for static/media; optional CDN (Cloud CDN/CloudFlare).
* Cloud Scheduler + Cloud Tasks for background jobs.
* **Networking**:
* VPC with connector for Cloud Run → Cloud SQL.
* Restrict ingress; enforce HTTPS; custom domain + managed certs.
* **IAM**:
* Least privilege service accounts for Cloud Run and tasks.

**12) CI/CD and environments**

* **Branches**: main, staging, dev.
* **Cloud Build**:
* Build Docker image, run tests, run Alembic migrations, deploy to Cloud Run.
* Separate configs for staging/prod; manual approval for prod.
* **Database migrations**: Auto-run alembic upgrade head during deploy with lock to prevent parallel runs.
* **Smoke tests**: Healthcheck endpoint, basic booking flow in staging.

**13) Observability and reliability**

* **Logging**: Structured JSON logs; request IDs; include park\_id.
* **Metrics**: Requests, booking throughput, payment success rate, availability generation time.
* **Tracing**: OpenTelemetry to Cloud Trace.
* **Error tracking**: Sentry or Error Reporting.
* **Backups**: Automated Cloud SQL backups and PITR; tested restore plan.

**14) Security and compliance**

* **App**: CSRF, XSS, SSRF protections; input validation; strict CSP; secure cookies.
* **Payments**: Avoid storing PAN; use Stripe tokens; PCI scope minimized.
* **PII**: Data retention policy; encrypted at rest (Cloud SQL) and in transit; field-level encryption if needed.
* **RBAC**: Per-park isolation in queries; enforce on every admin route; audit log.
* **Rate limiting**: Per-IP and per-route (login, webhooks).

**15) Testing strategy**

* **Unit tests**: services (availability, pricing, payments).
* **Integration tests**: booking flow with test DB; Stripe test keys + webhook simulation.
* **E2E**: Happy path customer booking; admin refund; calendar drag-drop.
* **Load testing**: Peak booking windows; evaluate slot contention correctness.
* **Security tests**: Authorization boundaries across park\_id, webhook signature validation.

**16) Data migration and content import**

* **Designer assets**: Convert static HTML/CSS into Jinja templates; centralize layout/partials.
* **Initial content**: Seed demo park, fields, packages; migration scripts for computed columns/indexes.

**17) Launch checklist**

* **Functional acceptance**: All flows approved by 2–3 pilot parks.
* **SLA/SLO**: Error budget and alerting in place.
* **Runbook**: Incident response, rollback steps, on-call.
* **Cost review**: Cloud SQL size, Cloud Run min instances, egress/CDN.

**Bullet point list of steps (quick checklist)**

* **Plan**: Roles, flows, tenancy, non-functional requirements.
* **Design**: ERD, APIs, blueprints, template map.
* **Setup**: Repo, venv, dependencies, base Flask app factory.
* **DB**: SQLAlchemy models, Alembic migrations, local Postgres.
* **Auth**: Flask-Login, RBAC by park\_id, CSRF.
* **Availability**: Slot generation, capacity rules, contention-safe reservation.
* **Pricing**: Base + addons + coupons + taxes; service layer.
* **Payments**: Stripe PaymentIntents, webhooks, idempotency, refunds.
* **Customer UI**: Landing, calendar, booking form, checkout, confirmation.
* **Admin UI**: Dashboard, drag-drop calendar, config, reports, refunds.
* **Emails**: Confirmation, reminder, cancellation; per-park branding.
* **Background jobs**: Hold expiry, reconciliation, reminders (Scheduler/Tasks).
* **Infra**: Cloud Run, Cloud SQL, VPC, Secret Manager, Storage bucket/CDN.
* **CI/CD**: Cloud Build pipelines, migrations, deploy to staging/prod.
* **Observability**: Logs, metrics, tracing, error tracking, alerts.
* **Security**: CSP, CSRF, input validation, rate limiting, audit logs, backups.
* **Testing**: Unit/integration/E2E/load; webhook simulation.
* **Go-live**: Domain + SSL, runbook, pilot validation, cost tuning.

**Suggested next steps**

* I can generate the initial Flask project structure, dependencies, and starter templates tailored to this stack. If you want, say “bootstrap the repo” and I’ll scaffold it with config, models, blueprints, and CI/CD stubs.

**Short, high‑level plan I can send the client**

* **Phase 0 – Kickoff & alignment**
* Confirm goals, user roles, and MVP scope
* Approve GCP resources and access
* Agree on weekly update cadence and channels
* **Phase 1 – Architecture & setup**
* Define data model and app blueprints
* Set up repo, environments, and CI/CD
* Provision core GCP infra
* **Phase 2 – Core booking flow**
* Availability generation and capacity rules
* Booking creation, holds, and confirmations
* Basic customer-facing pages (browse → select → book)
* **Phase 3 – Payments**
* Integrate Stripe PaymentIntents + webhooks
* Confirmation emails/receipts and refunds
* **Phase 4 – Admin console**
* Calendar management, manual bookings, blackout dates
* Park configuration (fields, packages, pricing)
* **Phase 5 – QA, security, and launch**
* End-to-end tests, monitoring, error handling
* Staging review, production cutover, runbook

**GCP resources we plan to use**

* **Compute**: Cloud Run (containerized Flask app)
* **Database**: Cloud SQL for PostgreSQL (Primary + automated backups)
* **Networking**: VPC + Serverless VPC Access connector (private DB access)
* **Secrets**: Secret Manager (Stripe keys, DB creds)
* **Storage/CDN**: Cloud Storage bucket (static/media), optional Cloud CDN
* **Build/Deploy**: Cloud Build + Artifact Registry (images), Git-triggered pipelines
* **Scheduling/Async**: Cloud Scheduler + Cloud Tasks (reminders, hold expiry)
* **Observability**: Cloud Logging, Cloud Monitoring (metrics/alerts), Error Reporting, Cloud Trace
* **DNS/SSL**: Cloud DNS (if managing domain) + Managed SSL certs on Cloud Run
* **IAM/Security**: Least-privilege service accounts, per-environment separation
* Optional later: Memorystore (Redis) for caching if needed

**Update cadence**

* **Twice-weekly updates**: concise progress + what’s next + risks
* Example: Tuesdays and Fridays via email/Slack with a running changelog
* **Live artifacts**: staging URL for demos, CI/CD build logs, and a simple status board
* **Ad-hoc**: immediate note for any blockers or critical decisions

**Next steps to start**

* Confirm the GCP resources list above
* Share or set up the GCP project and access
* Approve the update cadence and communication channel
* I’ll provision the baseline infra and push the initial app scaffolding for review within the first week

If this aligns, I’ll send a one‑page plan and the initial milestone schedule for approval.