E-commerce Platform — Step-by-Step Plan & Milestones

This plan breaks the project into small, shippable steps. Each step lists: **Goal → Tasks → Deliverables → Definition of Done (DoD)**. We'll iterate after each step.

> Tech baseline assumed: Go 1.22+, gRPC, Buf for protobufs, Docker, Postgres, OpenTelemetry, Prometheus/Grafana. Replace tools if you prefer.

Step 0 — Repo bootstrap & developer ergonomics

Goal: One repo that builds locally with minimal friction.

Tasks - Initialize Git repo; add `.editorconfig`, `.gitignore`, `README.md` skeleton. - Create `go.work` at repo root to tie all modules together (api-gateway, user-service, product-service, order-service, payment-service). - Create `Makefile` with common targets: - `make dev` (docker compose up), `make down`, `make logs`, `make lint`, `make test`, `make proto`, `make run-<svc>`, `make e2e`. - Add `tools/` (or `internal/tools`) module pinning CLIs (buf, protoc-gen-go, golangci-lint, migrate) via `go install` recipes.

Deliverables - Working `go.work` and Makefile. - Pre-commit hooks (format, lint).

DoD - `make lint` and `make test` run successfully (even if no tests yet).

Step 1 — Protobuf contracts & codegen

Goal: Single source of truth for APIs with automated codegen.

Tasks - Keep shared contracts in `/proto` (user.proto, product.proto, order.proto, payment.proto). - Add `buf.yaml` + `buf.gen.yaml` with lint and breaking change checks. - Generate Go stubs into each service under `internal/gen` (or `/gen`). - Remove per-service duplicated `proto/` copies; replace with generated imports.

Deliverables - `buf` configured, `make proto` regenerates stubs deterministically.

DoD - `make proto` succeeds and gRPC packages compile in all services.

Step 2 — Shared libraries: config, logging, tracing, auth

Goal: Consistent cross-cutting infra.

Tasks - Create `internal/pkg/` (or `/pkg`) module with: - Config loader (env + file) with structs per service. - Logger (zerolog or slog), request IDs. - OpenTelemetry setup (OTLP exporter); HTTP & gRPC interceptors to propagate trace context. - JWT utilities (RS256), key rotation support.

Deliverables - Reusable packages imported by gateway and services.

DoD - Each binary starts with consistent logging + tracing initialization.

Step 3 — Local dev stack (docker-compose)

Goal: One command to get infra up.

Tasks - `deployments/docker-compose.yml` with: - Postgres containers (separate DBs for user, product, order). Optionally Redis for caching/sessions. - NATS or Kafka (optional now, used later for events). - Prometheus, Grafana, and Tempo/Jaeger for traces. - Add `scripts/wait-for.sh` for startup ordering. - Add `deployments/migrations/` and wire golang-migrate per service.

Deliverables - `make dev` brings up infra; `make down` tears it down.

DoD - Containers are healthy; services can connect to their DBs.

Step 4 — API Gateway skeleton (REST → gRPC)

Goal: Public HTTP entrypoint with health and routing skeleton.

Tasks - `api-gateway` starts HTTP server with middleware: logging, recovery, CORS, JWT (optional for now), request ID, trace injection. - Define `/healthz` and `/readyz`. - Add minimal routes that proxy to gRPC (even if services respond with UNIMPLEMENTED).

Deliverables - Running gateway with health endpoints.

DoD - `curl :8080/healthz` returns 200; traces visible in Tempo/Jaeger.

Step 5 — User Service MVP (auth + profiles)

Goal: Users can sign up, log in, and fetch profile.

Tasks - Schema/migrations: `users(id, email unique, password_hash, created_at, updated_at)`. - gRPC methods: `Register`, `Login`, `GetProfile`. - Password hashing (argon2id or bcrypt), email normalization. - JWT issuance (access + refresh) with RS256; store refresh token family. - Gateway REST endpoints mapping to gRPC. - Basic validation and error model.

Deliverables - `POST /v1/auth/register`, `POST /v1/auth/login`, `GET /v1/me`.

DoD - Manual flow works via curl/Postman; user rows appear in DB; tokens validate.

Step 6 — Product Service MVP (catalog + inventory)

Goal: List products and manage stock.

Tasks - Schema: `products(id, sku unique, name, description, price_cents, currency, stock, created_at, updated_at)`. - gRPC: `CreateProduct`, `ListProducts`, `GetProduct`, `UpdateProduct`, `AdjustStock`. - Gateway REST: `/v1/products` (GET/POST), `/v1/products/:id` (GET/PATCH), auth required for write. - Optional read cache (Redis) for list endpoints.

Deliverables - Seed script `scripts/seed_products.go`.

DoD - `GET /v1/products` returns seeded items; metrics exported.

Step 7 — Order Service MVP (orders only)

Goal: Create orders and manage status (no payments yet).

Tasks - Schema: `orders(id, user_id, status, total_cents, currency, created_at)` + `order_items(order_id, product_id, quantity, price_cents)`. - States: `PENDING → CONFIRMED → CANCELLED` (payments will confirm later). - On create: compute totals from product service; (option A) sync call to decrement stock; (option B) reserve then confirm. - gRPC: `CreateOrder`, `GetOrder`, `ListOrders`, `UpdateOrderStatus`. - REST mapping via gateway.

Deliverables - Endpoints to create and inspect orders; basic stock handling.

DoD - End-to-end: register → login → list products → create order (1 item) → inspect.

Step 8 — Payment Service (stub → Stripe/PayPal later)

Goal: Pluggable payments interface; start with simulator.

Tasks - Contract: `CreatePaymentIntent(order_id, amount, currency)`, `GetPaymentStatus`. - Simulator: randomly succeed/fail with deterministic seed in dev. - Order service reacts to payment success/failure (sync now; async later via events).

Deliverables - `/v1/orders/:id/pay` endpoint through gateway.

DoD - Happy path: order moves to `CONFIRMED` on simulated success and to `CANCELLED` on failure; stock restored on failure.

Step 9 — Observability & resilience hardening

Goal: Production-ready signals + failure tolerance.

Tasks - OpenTelemetry spans across gateway↔services; trace IDs in logs. - Prometheus counters/histograms for requests, latencies, errors. - Grafana dashboards per service. - Timeouts, retries with backoff, and circuit breakers for inter-service calls. - Structured error model (gRPC status + details → REST problem+json).

Deliverables - Dashboards JSON in `/deployments/grafana/`; Helm or k8s manifests updated if applicable.

DoD - Load test shows traces and sensible metrics; retry/breaker behavior validated.

Step 10 — CI/CD

Goal: Automated checks and artifacts per commit.

Tasks - GitHub Actions workflow: - Lint, vet, unit tests. - `buf lint` + `buf breaking`. - `make proto` and ensure no diff. - Build Docker images; push to registry on main tags. - Optional: e2e with `docker-compose` in CI. - Cache Go build and Docker layers for speed.

Deliverables - `.github/workflows/ci-cd.yml` with the above jobs.

DoD - PRs show green checks; artifacts/images available.

Step 11 — Kubernetes (optional now, ready later)

Goal: Declarative deploys and scaling.

Tasks - Manifests under `deployments/k8s/` for gateway + each service + Prometheus/Grafana. - Use secrets for JWT keys and DB creds (sealed-secrets or external secret manager). - PodDisruptionBudgets, HPAs, liveness/readiness probes.

Deliverables - Working manifests or Helm charts.

DoD - Cluster comes up with healthy pods; traffic flows through gateway svc/ingress.

Step 12 — E2E tests & smoke suite

Goal: Repeatable end-to-end verification.

Tasks - Add Go e2e tests (or Postman collection + Newman) that run against local stack. - Scenarios: - Register → Login → List Products → Create Order → Pay (success) → Verify. - Insufficient stock path. - Payment failure rollback. - Seed fixtures and clean teardown.

Deliverables - `make e2e` target; reports saved under `artifacts/`.

DoD - E2E suite passes locally and in CI.

Cutlines / Increments

- **MVP 1 (Steps 0–6):** Browse catalog with auth.
- **MVP 2 (Steps 0–8):** Place and pay for orders (simulated).
- **Prod-ready (Steps 0–10 + 12):** Observability + CI/CD + tests.

Suggested folder tweaks

- Keep shared libs in `internal/pkg` or `pkg/` to avoid circular deps.
- Generate protobuf into `internal/gen` to keep APIs private to each module.
- Consider `/docs/` for API examples and an `openapi.yaml` exported from the gateway if you add grpc-gateway/Swagger later.

Reference Makefile targets (sketch)

```
proto: ## generate protobuf stubs
    buf generate

lint: ## static checks
    golangci-lint run ./...

test: ## unit tests
    go test ./...

dev: ## local stack up
    docker compose -f deployments/docker-compose.yml up -d

down:
    docker compose -f deployments/docker-compose.yml down -v

run-%: ## run a service (e.g., make run-user-service)
    cd $* && go run ./cmd/$*

e2e:
    go test ./tests/e2e -v
```

Acceptance examples (curl)

- Health: `curl :8080/healthz`
- Register: `curl -XPOST :8080/v1/auth/register -d '{"email":"a@b.co","password":"P@ssw0rd"}' -H 'Content-Type: application/json'`
- List products: `curl :8080/v1/products`

- Create order:
`curl -XPOST :8080/v1/orders -H 'Authorization: Bearer <token>' -d '{"items":[{"product_id":"<id>","qty":1}]}'`

Decision & risk log (living)

- **Stock reservation strategy:** immediate decrement vs. reservation window — choose by Step 7.
- **Async messaging:** NATS/Kafka introduced post-MVP if you need decoupling.
- **Payments:** start with simulator; swap to Stripe/PayPal adapters later.

What “we have” after each key step

- **After Step 4:** A running gateway with health checks and tracing.
- **After Step 5:** Secure auth with JWT; users can log in.
- **After Step 6:** Browsable catalog; seed data visible via REST.
- **After Step 7:** Orders without payment; totals computed; stock updated.
- **After Step 8:** End-to-end purchase flow (simulated payments).
- **After Step 9–10:** Metrics, traces, and CI/CD safety net.
- **After Step 12:** Repeatable e2e confidence on every change.

Next suggested move: Start with Step 0–1. I can draft the `go.work`, `buf.yaml`, `buf.gen.yaml`, and a Makefile skeleton when you’re ready.