✅Next Steps: Harden Monorepo

Below are the next enhancements to implement, building on the scaffold:

---

1. Database Integration

- Implement real DB queries in `handlers/signals.rs` and `handlers/backtests.rs`.
- Add SQLx query macros for fetching signals and storing backtest results.
- Create TimescaleDB hypertables for time-series signals.

2. Authentication & Authorization

- Add JWT middleware using Supabase public key (JWK verification).
- Create middleware for RBAC: analyst, ops, compliance.
- Protect `/v1/portfolio/publish` and `/v1/compliance` routes with roles.

3. Agent Integration Hooks

- Replace demo logic with async requests to `signal_svc`, `backtest_svc`, etc.
- Use message bus (NATS/Kafka) for delegation and A2A coordination.
- Implement MCP tool specs for Rust agents.

4. Infrastructure-as-Code

- Terraform for:
- PostgreSQL (RDS/CloudSQL or managed instance)
- S3-compatible object store
- Redis cluster for feature cache
- Kubernetes manifests:
- API Deployment + Service + Ingress (Traefik)
- StatefulSets for DB and feature store
- Helm charts for parameterization

5. CI/CD Expansion

- Add Rust test jobs (`cargo nextest`, property-based tests)
- Add Docker/Podman image build & push to container registry
- Deploy staging environment on PR merge using GitHub Actions

6. Observability Setup

- Add OpenTelemetry tracing for API and agents.
- Configure Prometheus + Grafana for metrics.

- Loki + Tempo for logs and traces.

7. Security Enhancements

- Enable HTTPS termination in Traefik with Let's Encrypt or managed certs.
- Secrets management via Vault + sealed secrets for Kubernetes.
- Image signing (Cosign) in CI pipeline.

8. Flutter Enhancements

- Add charts (e.g., `fl_chart`) for signals and performance metrics.
- Integrate Supabase Auth (email/password + social logins).
- Add persistent sessions and offline caching using `hive` or `isar`.
- Implement push updates via WebSockets or SSE for live signals.

Would you like me to **generate the full Terraform + Kubernetes manifests** next or **implement the actual DB-backed Rust endpoints with JWT auth**?

Backend Add-Ons — Auth (JWT / Supabase) + Real DB Wiring

New Dependencies

```
# backend/api/Cargo.toml (append)
jsonwebtoken = "9"
headers = "0.4"
reqwest = { version = "0.12", default-features = false, features = ["rustls-tls"] }
once_cell = "1"
axum-extra = { version = "0.9", features = ["typed-header"] }
```

Auth Extractor & Layer

```
// backend/api/src/auth.rs
use axum::{http::StatusCode, response::IntoResponse, RequestPartsExt};
use axum_extra::typed_header::TypedHeader;
use headers::{authorization::Bearer, Authorization};
use jsonwebtoken::{decode, DecodingKey, Validation, Algorithm};
use serde::Deserialize;
use std::sync::Arc;
use once_cell::sync::OnceCell;
use crate::state::AppState;
```

```rust
#[derive(Debug, Clone)]
pub struct AuthedUser { pub sub: String, pub email: Option<String>, pub role:
Option<String> }

#[derive(Debug, Deserialize)]
struct Claims { sub: String, email: Option<String>, role: Option<String>, exp:
usize, iss: Option<String>, aud: Option<String> }

static DECODING: OnceCell<DecodingKey> = OnceCell::new();

fn get_key() -> &'static DecodingKey {
    DECODING.get_or_init(|| {
        let secret =
std::env::var("SUPABASE_JWT_SECRET").expect("SUPABASE_JWT_SECRET not set");
        DecodingKey::from_secret(secret.as_bytes())
    })
}

pub async fn require_auth<B>(mut req: axum::http::Request<B>, next:
axum::middleware::Next<B>) -> impl IntoResponse {
    let TypedHeader(Authorization(bearer)) = match
req.extract::<TypedHeader<Authorization<Bearer>>>().await {
        Ok(h) => h,
        Err(_) => return (StatusCode::UNAUTHORIZED, "missing bearer
token").into_response(),
    };
    let token = bearer.token();
    let mut val = Validation::new(Algorithm::HS256);
    if let Ok(aud) = std::env::var("JWT_AUDIENCE") { val.set_audience(&[aud]); }
    if let Ok(iss) = std::env::var("JWT_ISSUER") { val.set_issuer(&[iss]); }
    match decode::<Claims>(token, get_key(), &val) {
        Ok(data) => {
            let user = AuthedUser { sub: data.claims.sub, email:
data.claims.email, role: data.claims.role };
            req.extensions_mut().insert(user);
            next.run(req).await
        }
        Err(_) => (StatusCode::UNAUTHORIZED, "invalid token").into_response(),
    }
}
```

Wire Middleware (protect everything except `/health` + docs)

```rust
// backend/api/src/main.rs (add at top)
mod routes; mod handlers; mod state; mod error; mod openapi; mod auth;
```

```
// backend/api/src/main.rs (build the app)
let protected =
routes::router(&state).layer(axum::middleware::from_fn(auth::require_auth));
let app = Router::new()
    .route("/health", get(handlers::health::get_health))
    .merge(SwaggerUi::new("/docs").url("/api-docs/openapi.json", openapi))
    .nest("/", protected);
```

Update OpenAPI Security

```
// backend/api/src/openapi.rs (append derive attributes if desired)
// utoipa supports security annotations per path; for brevity, document global
bearer scheme in README.
```

Real DB Query for `/v1/signals`

```
// backend/api/src/handlers/signals.rs
use axum::{Json, extract::{State, Query}};
use serde::Deserialize;
use chrono::Utc;
use uuid::Uuid;
use common::Signal;
use crate::state::AppState;
use sqlx::FromRow;

#[derive(Deserialize)]
pub struct SignalQuery { pub symbol: Option<String>, pub horizon:
Option<String>, pub limit: Option<i64> }

#[derive(FromRow)]
struct DbSignal { id: Uuid, asof: chrono::DateTime<Utc>, symbol: String,
model_version: String, horizon: String, score: f64, confidence: f64, explain:
serde_json::Value }

#[utoipa::path(get, path="/v1/signals", params(("symbol" = Option<String>,
Query),("horizon" = Option<String>, Query),("limit" = Option<i64>, Query)),
responses((status=200, body=[Signal])))]
pub async fn get_signals(State(st): State<AppState>, Query(q):
Query<SignalQuery>) -> Json<Vec<Signal>> {
    let limit = q.limit.unwrap_or(20).min(200);
    let rows: Vec<DbSignal> = if let Some(sym) = &q.symbol {
        if let Some(h) = &q.horizon {
            sqlx::query_as::<_, DbSignal>(
                "SELECT id, asof, symbol, model_version, horizon, score,
```

```
confidence, explain FROM signals WHERE symbol = $1 AND horizon = $2 ORDER BY
asof DESC LIMIT $3"
            ).bind(sym).bind(h).bind(limit).fetch_all(&st.db).await.unwrap()
        } else {
            sqlx::query_as::<_, DbSignal>(
                "SELECT id, asof, symbol, model_version, horizon, score,
confidence, explain FROM signals WHERE symbol = $1 ORDER BY asof DESC LIMIT $2"
            ).bind(sym).bind(limit).fetch_all(&st.db).await.unwrap()
        }
    } else {
        sqlx::query_as::<_, DbSignal>(

"SELECT id, asof, symbol, model_version, horizon, score, confidence, explain
FROM signals ORDER BY asof DESC LIMIT $1"
        ).bind(limit).fetch_all(&st.db).await.unwrap()
    };

    let out = rows.into_iter().map(|r| Signal { asof: r.asof, symbol: r.symbol,
signal_id: r.id, model_version: r.model_version, horizon: r.horizon, score:
r.score, confidence: r.confidence, explain: r.explain }).collect();
    Json(out)
}
```

Persist `/v1/backtests/run`

```
-- backend/api/migrations/0002_backtests.sql
CREATE TABLE IF NOT EXISTS backtests (
  run_id UUID PRIMARY KEY,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  strategy_id TEXT NOT NULL,
  start_ts TIMESTAMPTZ NOT NULL,
  end_ts TIMESTAMPTZ NOT NULL,
  sharpe DOUBLE PRECISION NOT NULL,
  max_dd DOUBLE PRECISION NOT NULL,
  turnover DOUBLE PRECISION NOT NULL,
  summary JSONB NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_backtests_strategy_time ON
backtests(strategy_id, created_at DESC);
```

```
// backend/api/src/handlers/backtests.rs
use axum::{Json, extract::State};
use uuid::Uuid;
use common::{BacktestRequest, BacktestReport};
use crate::state::AppState;
```

```
#[utoipa::path(post, path="/v1/backtests/run", request_body=BacktestRequest,
responses((status=200, body=BacktestReport)))]
pub async fn post_run_backtest(State(st): State<AppState>, Json(req):
Json<BacktestRequest>) -> Json<BacktestReport> {
    // TODO: delegate to backtest_svc; for now compute simple demo metrics
deterministically from inputs
    let run_id = Uuid::new_v4();
    let days = ((req.end - req.start).num_days().max(1)) as f64;
    let sharpe = (1.0 + (req.costs_bps.unwrap_or(5.0) * -0.0001)).max(0.1);
    let max_dd = -0.08_f64;
    let turnover = 0.35_f64;
    let report = BacktestReport { run_id, sharpe, max_dd, turnover, summary:
serde_json::json!({
        "strategy_id": req.strategy_id,
        "period_days": days,
        "note": "demo; replace with backtest_svc results"
    }) };

    sqlx::query!(
        "INSERT INTO backtests(run_id, strategy_id, start_ts, end_ts, sharpe,
max_dd, turnover, summary) VALUES($1,$2,$3,$4,$5,$6,$7,$8)",
        report.run_id, req.strategy_id, req.start, req.end, report.sharpe,
report.max_dd, report.turnover, report.summary
    ).execute(&st.db).await.unwrap();

    Json(report)
}
```

Environment Variables (backend)

```
# backend/.env.example (append)
SUPABASE_JWT_SECRET=replace_me
JWT_AUDIENCE=authenticated
JWT_ISSUER=https://YOUR.supabase.co/auth/v1
```

Frontend Add-Ons — Supabase Auth & Bearer Token to API

Update `lib/services/api_service.dart`

```
// attach bearer from Supabase session automatically
import 'package:supabase_flutter/supabase_flutter.dart';
...
```

```
  Map<String, String> _headers(){
    final token = Supabase.instance.client.auth.currentSession?.accessToken;
    final h = {'Content-Type': 'application/json'};
    if (token != null) h['Authorization'] = 'Bearer $token';
    return h;
  }

  Future<List<SignalModel>> fetchSignals({String symbol = 'AAPL', String
horizon = '1d', int limit = 10}) async {
    final uri = Uri.parse('$baseUrl/v1/signals?
symbol=$symbol&horizon=$horizon&limit=$limit');
    final res = await http.get(uri, headers: _headers());
    if (res.statusCode == 401) { throw Exception('Unauthorized — sign in'); }
    if (res.statusCode != 200) throw Exception('Failed to fetch signals');
    ...
  }

  Future<Map<String, dynamic>> pretradeCheck(Map<String, dynamic> body) async {
    final uri = Uri.parse('$baseUrl/v1/compliance/pretrade-check');
    final res = await http.post(uri, body: json.encode(body), headers:
_headers());
    if (res.statusCode == 401) { throw Exception('Unauthorized — sign in'); }
    if (res.statusCode != 200) throw Exception('Compliance check failed');
    return json.decode(res.body) as Map<String, dynamic>;
  }
```

Update `lib/features/auth/login_page.dart`

```
import 'package:supabase_flutter/supabase_flutter.dart';
...
FilledButton(onPressed:() async {
  try {
    await Supabase.instance.client.auth.signInWithPassword(email: _email.text,
password: _pw.text);
    if (context.mounted) context.go('/signals');
  } catch (e) {
    if (context.mounted)
ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Login
failed: $e')));
  }
}, child: const Text('Sign in')),
```

Initialize Supabase in `main.dart`

```
import 'services/auth_service.dart';
...
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await AuthService.init(url: const String.fromEnvironment('SUPABASE_URL',
defaultValue: ''), anonKey: const String.fromEnvironment('SUPABASE_ANON_KEY',
defaultValue: ''));
  runApp(const ProviderScope(child: MarketIntelApp()));
}
```

Kubernetes Manifests (prod-ready skeleton)

```
# infra/k8s/namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: mai
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-config
  namespace: mai
data:
  BIND_ADDR: "0.0.0.0:8080"
  RUST_LOG: "api=info"
  DATABASE_URL: "postgres://postgres:postgres@db:5432/mai"
---
apiVersion: v1
kind: Secret
metadata:
  name: api-secrets
  namespace: mai
type: Opaque
stringData:
  SUPABASE_JWT_SECRET: "replace_me"
  JWT_AUDIENCE: "authenticated"
  JWT_ISSUER: "https://YOUR.supabase.co/auth/v1"
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: api
  namespace: mai
spec:
  replicas: 2
  selector: { matchLabels: { app: api } }
  template:
    metadata: { labels: { app: api } }
    spec:
      containers:
      - name: api
        image: ghcr.io/YOURORG/mai-api:latest
        ports: [{ containerPort: 8080 }]
        envFrom:
        - configMapRef: { name: api-config }
        - secretRef: { name: api-secrets }
        readinessProbe: { httpGet: { path: /health, port: 8080 },
initialDelaySeconds: 5, periodSeconds: 10 }
        livenessProbe: { httpGet: { path: /health, port: 8080 },
initialDelaySeconds: 10, periodSeconds: 20 }
        resources: { requests: { cpu: "200m", memory: "256Mi" }, limits: { cpu:
"1", memory: "1Gi" } }
---
apiVersion: v1
kind: Service
metadata:
  name: api
  namespace: mai
spec:
  selector: { app: api }
  ports: [{ name: http, port: 80, targetPort: 8080 }]
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
  namespace: mai
spec:
  scaleTargetRef: { apiVersion: apps/v1, kind: Deployment, name: api }
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: { type: Utilization, averageUtilization: 70 }
```

Note: Use a managed Postgres (RDS/CloudSQL) for production; in-cluster DBs are for dev only.

Terraform (cluster-agnostic, assumes an existing K8s cluster via KUBECONFIG)

```
# infra/terraform/main.tf
terraform {
  required_version = ">= 1.6.0"
  required_providers {
    kubernetes = { source = "hashicorp/kubernetes", version = ">= 2.27.0" }
    helm       = { source = "hashicorp/helm",       version = ">= 2.13.1" }
  }
}

provider "kubernetes" {}
provider "helm" {}

variable "namespace" { default = "mai" }
variable "supabase_jwt_secret" { sensitive = true }
variable "database_url" { description = "Postgres URL" }

resource "kubernetes_namespace" "mai" { metadata { name = var.namespace } }

resource "kubernetes_config_map" "api" {
  metadata { name = "api-config" namespace = var.namespace }
  data = { BIND_ADDR = "0.0.0.0:8080", RUST_LOG = "api=info", DATABASE_URL =
var.database_url }
}

resource "kubernetes_secret" "api" {
  metadata { name = "api-secrets" namespace = var.namespace }
  data = {
    SUPABASE_JWT_SECRET = base64encode(var.supabase_jwt_secret)
    JWT_AUDIENCE        = base64encode("authenticated")
    JWT_ISSUER          = base64encode("https://YOUR.supabase.co/auth/v1")
  }
}

resource "kubernetes_deployment" "api" {
  metadata { name = "api" namespace = var.namespace labels = { app = "api" } }
  spec {
    replicas = 2
    selector { match_labels = { app = "api" } }
    template {
```

```
      metadata { labels = { app = "api" } }
      spec {
        container {
          name  = "api"
          image = "ghcr.io/YOURORG/mai-api:latest"
          port { container_port = 8080 }
          env_from { config_map_ref { name =
kubernetes_config_map.api.metadata[0].name } }
          env_from { secret_ref     { name =
kubernetes_secret.api.metadata[0].name } }
        }
      }
    }
  }
}

resource "kubernetes_service" "api" {
  metadata { name = "api" namespace = var.namespace }
  spec {
    selector = { app = "api" }
    port { name = "http" port = 80 target_port = 8080 }
  }
}
```

Terraform usage

```
cd infra/terraform
terraform init
terraform apply -var="database_url=postgres://..." -var="supabase_jwt_secret=$
(op read ...)"
```

README Updates — Auth & Deployment

```
### Auth
- Set `SUPABASE_JWT_SECRET` in backend env (from Supabase project settings → API
→ JWT secret).
- Frontend uses Supabase session token as `Authorization: Bearer <token>` for
API routes.

### DB
- Run migrations automatically at startup; for new tables, add files under
`backend/api/migrations/`.
```

```
### K8s
- Build & push image: `podman build -t ghcr.io/YOURORG/mai-api:latest backend/
api && podman push ghcr.io/YOURORG/mai-api:latest`.
- Apply manifests: `kubectl apply -f infra/k8s/`.

### Terraform
- Assumes an existing cluster and context; installs ConfigMap/Secret/Deployment/
Service.
```

Next Engineering Steps

- Replace demo backtest in `post_run_backtest` with a call to `backtest_svc` and persist its results; emit WORM audit log.
- Add RBAC based on JWT `role` claim (analyst/trader/compliance) with per-route guards.
- Implement `/v1/portfolio/publish` integration with optimizer + risk checks and pre-trade compliance.
- Add Observability: OpenTelemetry tracing, Prometheus metrics, and ServiceMonitor.
- Introduce Kafka/Redpanda for event bus; publish `SIGNAL`, `BACKTEST`, and `PORTFOLIO` events.