Repo layout

This layout assumes a monorepo with shared infra, agents, and per-service apps.

```text
phase1-ecosystem/
  README.md

  config/
    env.global.example.yaml
    cache_key_conventions.yaml
    dragonfly-cluster.yaml

    hyperlift/
      creditx-service.yaml
      threat-service.yaml
      guardian-service.yaml
      apps-service.yaml
      phones-service.yaml

    agents/
      orchestrator_prompt.yaml
      recovery_agent_prompt.yaml
      agent_registry.yaml

    monitoring/
      prometheus-rules.yaml
      grafana-notes.md

  services/
    shared/
      python/
        core_cache.py
      node/
        core_cache.ts

    creditx-service/
      app/
        main.py
        db.py
        models.py
        routes_compliance.py
      requirements.txt
      Dockerfile

    threat-service/
      app/
        main.py
        ...
      requirements.txt
      Dockerfile
    guardian-service/
      app/
        main.py
        ...
      requirements.txt
      Dockerfile

    apps-service/
      src/
        app.ts
        routes.ts
      package.json
      tsconfig.json
      Dockerfile

    phones-service/
      src/
        app.ts
        routes.ts
      package.json
      tsconfig.json
      Dockerfile

  .github/
    workflows/
      deploy-creditx.yaml
      deploy-apps.yaml
      deploy-threat.yaml
      deploy-guardian.yaml
      deploy-phones.yaml
```

Core config & infra files

These files capture the environment contract, Dragonfly cluster, and cache mapping.

`config/env.global.example.yaml`

```text
# Common ENV (injected via Hyperlift)

# Core infrastructure
SPACES_ENV: "production"
SPACES_REGION: "us-phx-1"

# Database
DATABASE_URL: "postgresql+psycopg2://
ecosystem:***@postgres.internal:5432/ecosystem"

# Cache (Dragonfly)
CACHE_HOST: "dragonfly-cache.internal"
CACHE_PORT: "6379"
CACHE_SSL: "false"
```

```
CACHE_DB_MAIN: "0"       # default; per-service overrides below
CACHE_TIMEOUT_MS: "30000"
CACHE_MAX_POOL_SIZE: "50"

# Observability
PROMETHEUS_ENDPOINT: "http://prometheus.internal:9090"
JAEGER_AGENT_HOST: "jaeger.internal"
JAEGER_AGENT_PORT: "6831"

# Agent mesh
AGENT_REGISTRY_URL: "http://agent-mesh.internal"
ORCHESTRATOR_URL: "http://orchestrator.internal"

# Security
JWT_PUBLIC_KEY: "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC
KEY-----"
OAUTH_ISSUER: "https://auth.ecosystem.ai"
```

config/cache_key_conventions.yaml

```text
cache_key_conventions:
  compliance_documents:
    key: "creditx:doc:{document_id}"
    ttl_sec: 604800          # 7 days

  automation_jobs:
    key: "apps:job:{job_id}"
    ttl_sec: 86400           # 1 day

  threat_events:
    key: "threat:event:{event_id}"
    ttl_sec: 900             # 15 minutes

  device_telemetry:
    key: "guardian:device:{device_id}"
    ttl_sec: 86400           # 1 day

  phone_locations:
    key: "phones:loc:{device_id}"
    ttl_sec: 86400           # 1 day
```

config/dragonfly-cluster.yaml

```text
dragonfly-cluster:
  vm:
    name: cache-prod-01
    tier: Standard-3
    cpu: 4
    ram_gb: 8
    storage_volume:
      name: dragonfly-cache-volume
      size_gb: 100
      mount_path: /mnt/volumes/dragonfly
      encrypted: true        # AES-256
  container:
    image: "dragonflydb/dragonfly:latest"
    ports:
      - 6379
    env:
      DFLY_aof_fsync_sec: 1
      DFLY_max_memory_policy: "allkeys_lru"
    restart_policy: "unless-stopped"
  db_mapping:
    creditx-service:
      db: 0
      prefix: "creditx:"
    threat-service:
      db: 1
      prefix: "threat:"
    guardian-service:
      db: 2
      prefix: "guardian:"
    apps-service:
      db: 3
      prefix: "apps:"
    phones-service:
      db: 4
      prefix: "phones:"
  monitoring:
    prometheus_exporter: true
    alerts:
      - rule: "cache_mem_usage > 0.8"
      - rule: "cache_latency_p95_ms > 10"
```

Hyperlift service manifests (Spaceship-native)

These capture the Spaceship Starlight + Hyperlift deployment for each service from GitHub + Dockerfile.

`config/hyperlift/creditx-service.yaml`

```
text
apiVersion: spaceship/v1
kind: Service
metadata:
  name: creditx-service
  namespace: ecosystem-prod
spec:
  source:
    repo: git@github.com:your-org/creditx-service.git
    branch: main
    dockerfile: ./Dockerfile
  image:
    repository: spaceship.registry.io/ecosystem/creditx-service
    tag: v2.0.0-dragonfly
  replicas:
    min: 3
    max: 10
    target_cpu_utilization: 70
  env:
    - name: SPACES_ENV
      value: "production"
    - name: DATABASE_URL
      valueFrom: secretRef:
        name: ecosystem-db
        key: DATABASE_URL
    - name: CACHE_HOST
      value: "dragonfly-cache.internal"
    - name: CACHE_PORT
      value: "6379"
    - name: CACHE_DB
      value: "0"
    - name: CACHE_KEY_PREFIX
      value: "creditx:"
    - name: CACHE_MAX_POOL_SIZE
      value: "50"
  health_checks:
    liveness:
      http_get:
        path: /health/live
        port: 8000
    readiness:
      http_get:
        path: /health/ready
        port: 8000
  update_strategy:
    type: rolling_update
    max_surge: 1
    max_unavailable: 0
  guards:
    error_rate_threshold: 0.01
    latency_p95_ms_threshold: 2000
    auto_rollback: true
```

You can clone this manifest for Threat, Guardian, Apps, and Phones, adjusting `CACHE_DB`, `CACHE_KEY_PREFIX`, and repo details.

Shared cache libraries (Python & Node)

Dragonfly is exposed through Redis-protocol endpoints, so existing Redis clients (`aioredis`, `ioredis`) work with no code-level API change, while benefiting from much higher throughput and lower tail latency.

services/shared/python/core_cache.py

```python
import aioredis
import json
import logging
import os

logger = logging.getLogger(__name__)

class DragonflyCache:
    def __init__(self):
        host = os.getenv("CACHE_HOST", "dragonfly-cache.internal")
        port = os.getenv("CACHE_PORT", "6379")
        db = int(os.getenv("CACHE_DB", "0"))
        self.prefix = os.getenv("CACHE_KEY_PREFIX", "")
        self._dsn = f"redis://{host}:{port}/{db}"
        self._pool = None

    async def connect(self):
        if self._pool is None:
            self._pool = await aioredis.create_redis_pool(
                self._dsn,
                maxsize=int(os.getenv("CACHE_MAX_POOL_SIZE", "50")),
            )
            logger.info("Connected to Dragonfly cache at %s", self._dsn)

    async def close(self):
        if self._pool:
            self._pool.close()
            await self._pool.wait_closed()

    def _k(self, key: str) -> str:
        return f"{self.prefix}{key}"

    async def get(self, key: str):
        try:
            raw = await self._pool.get(self._k(key))
            if raw is None:
                return None
            return json.loads(raw)
        except Exception as e:
            logger.warning("Cache GET failed for %s: %s", key, e)
            return None

    async def set(self, key: str, value, ttl_sec: int = 3600):
        try:
            raw = json.dumps(value)
            await self._pool.setex(self._k(key), ttl_sec, raw)
        except Exception as e:
            logger.warning("Cache SET failed for %s: %s", key, e)

    async def delete(self, key: str):
        try:
            await self._pool.delete(self._k(key))
        except Exception as e:
            logger.warning("Cache DEL failed for %s: %s", key, e)

    async def cache_aside(self, key: str, ttl_sec: int, fetch_fn):
        cached = await self.get(key)
        if cached is not None:
            return cached
        value = await fetch_fn()
        await self.set(key, value, ttl_sec)
        return value
```
services/shared/node/core_cache.ts

```ts
import Redis from "ioredis";

const host = process.env.CACHE_HOST || "dragonfly-cache.internal";
const port = Number(process.env.CACHE_PORT || 6379);
const db   = Number(process.env.CACHE_DB || 3);
const prefix = process.env.CACHE_KEY_PREFIX || "apps:";

const client = new Redis({ host, port, db, lazyConnect: true });

client.on("error", (err) => {
  console.warn("Dragonfly cache error:", err.message);
});

export async function connectCache() {
  if (!client.status || client.status !== "ready") {
    await client.connect();
  }
}

function k(key: string) {
  return `${prefix}${key}`;
}

export async function cacheGet<T>(key: string): Promise<T | null> {
  try {
    const raw = await client.get(k(key));
    return raw ? (JSON.parse(raw) as T) : null;
  } catch (err) {
    console.warn("cacheGet failed:", err);
    return null;
  }
}

export async function cacheSet(key: string, value: any, ttlSec = 3600) {
  try {
    await client.set(k(key), JSON.stringify(value), "EX", ttlSec);
  } catch (err) {
    console.warn("cacheSet failed:", err);
  }
}
```

Example service: CreditX (Python/FastAPI)

This is a fully wired example; the other Python services can mirror this pattern, and Node services use the TypeScript cache wrapper.

services/creditx-service/app/db.py

```
python
import os
import databases

DATABASE_URL = os.getenv("DATABASE_URL")

database = databases.Database(DATABASE_URL)
```

services/creditx-service/app/models.py

```
python
from pydantic import BaseModel
from typing import Optional

class ComplianceDocument(BaseModel):
  document_id: str
  customer_id: str
  status: str
  payload: dict
  created_at: str
  updated_at: Optional[str]
```

services/creditx-service/app/routes_compliance.py

```
python
from fastapi import APIRouter, HTTPException
from .db import database
from .models import ComplianceDocument
from shared.python.core_cache import DragonflyCache
from pybreaker import CircuitBreaker, CircuitBreakerError
import logging

logger = logging.getLogger(__name__)

router = APIRouter(prefix="/creditx", tags=["creditx"])

cache = DragonflyCache()
cache_breaker = CircuitBreaker(fail_max=5, reset_timeout=60)
```

```
@cache_breaker
async def _safe_cache_get(key: str):
    return await cache.get(key)

async def get_with_guard(key: str, fetch_fn, ttl_sec: int):
    try:
        cached = await _safe_cache_get(key)
        if cached is not None:
            return cached
    except CircuitBreakerError:
        logger.warning("Cache circuit OPEN, falling back to DB for %s",
key)
        return await fetch_fn()

    value = await fetch_fn()
    await cache.set(key, value, ttl_sec)
    return value

@router.on_event("startup")
async def startup():
    await database.connect()
    await cache.connect()

@router.on_event("shutdown")
async def shutdown():
    await database.disconnect()
    await cache.close()

@router.get("/documents/{document_id}",
response_model=ComplianceDocument)
async def get_document(document_id: str):
    async def fetch():
        row = await database.fetch_one(
            "SELECT * FROM compliance_documents WHERE document_id
= :id",
            {"id": document_id},
        )
        if not row:
            raise HTTPException(status_code=404, detail="Document not
found")
        return dict(row)

    key = f"doc:{document_id}"
    result = await get_with_guard(key, fetch, ttl_sec=7 * 24 * 3600)
    return ComplianceDocument(**result)
```
services/creditx-service/app/main.py

```
python
import logging
from fastapi import FastAPI
from .routes_compliance import router as compliance_router

logging.basicConfig(level=logging.INFO)
```

```
app = FastAPI(title="CreditX Service", version="2.0.0-dragonfly")

@app.get("/health/live")
async def live():
    return {"status": "ok"}

@app.get("/health/ready")
async def ready():
    # In production you'd check DB + cache connectivity here
    return {"status": "ready"}

app.include_router(compliance_router)
```

services/creditx-service/requirements.txt

```
text
fastapi==0.115.0
uvicorn[standard]==0.30.0
databases[postgresql]==0.9.0
aioredis==2.0.1
pybreaker==1.0.1
pydantic==2.9.0
```

services/creditx-service/Dockerfile

```
text
FROM python:3.11-slim

WORKDIR /app

ENV PYTHONUNBUFFERED=1

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app ./app
COPY ../shared/python/core_cache.py ./shared/python/core_cache.py

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Apps service (Node.js) – example

services/apps-service/src/app.ts

```
ts
import express from "express";
import { connectCache, cacheGet, cacheSet } from "../../shared/node/
core_cache";

const app = express();
app.use(express.json());

app.get("/health/live", (_req, res) => res.json({ status: "ok" }));
app.get("/health/ready", (_req, res) => res.json({ status: "ready" }));

app.get("/jobs/:id", async (req, res) => {
  const id = req.params.id;
  const key = `job:${id}`;

  await connectCache();

  const cached = await cacheGet<any>(key);
  if (cached) {
    return res.json({ source: "cache", data: cached });
  }

  // TODO: Replace with real DB fetch
  const job = { job_id: id, status: "pending", payload: {} };

  await cacheSet(key, job, 86400);
  return res.json({ source: "db", data: job });
});

export default app;
```

Agent prompts, registry, and guards

These files encode your Phase 1 agent behavior with Dragonfly-aware semantics.

`config/agents/orchestrator_prompt.yaml`

```
text
system_prompt: |
  You are the Orchestration Agent for the Ecosystem platform.
  The platform is live in production on Spaceship Starlight with
Dragonfly as the
  shared in-memory cache layer.

  Your job is to coordinate domain agents (CreditX, 91 Apps, Global AI
Alert,
  Guardian AI, Stolen Phones) to complete workflows reliably.

  Core rules:
  - ALWAYS check dependency readiness (DB, Dragonfly, event bus) before
dispatching.
  - If Dragonfly is degraded, continue workflows using PostgreSQL and
mark
    results as "cache_degraded" for observability.
  - Use exponential backoff and circuit breakers to avoid cascading
failures.
  - Never drop a customer workflow silently; on repeated failure,
escalate to
    Recovery Agent and queue for manual review.
```

`config/agents/recovery_agent_prompt.yaml`

```text
task: |
  When called with a failure event, classify the failure as:
  - transient_network
  - dragonfly_degraded
  - database_issue
  - application_bug
  - configuration_error

  For dragonfly_degraded:
  - Open cache-related circuit breakers.
  - Instruct callers to use database-only code paths.
  - Schedule a health probe against Dragonfly every 30 seconds.
  - When 3 consecutive probes succeed with p95 latency < 5ms, allow
cache use again.
```

config/agents/agent_registry.yaml

```text
agent_registry:
  orchestrator-agent:
    url: http://orchestrator.internal
  recovery-agent:
    url: http://recovery.internal
  tuning-agent:
    url: http://tuning.internal

  creditx-compliance-agent:
    url: http://creditx-service.internal
    dependencies: [postgres, dragonfly]

  threat-agent:
    url: http://threat-service.internal
    dependencies: [postgres, dragonfly]

  guardian-agent:
    url: http://guardian-service.internal
    dependencies: [postgres, dragonfly]

  apps-agent:
    url: http://apps-service.internal
    dependencies: [postgres, dragonfly]

  phones-agent:
    url: http://phones-service.internal
    dependencies: [postgres, dragonfly]
```

CI/CD and deployment guards

Hyperlift still builds from GitHub + Dockerfile and handles blue-green cutovers; here it is wired with pre-flight checks to Dragonfly and readiness endpoints.

.github/workflows/deploy-creditx.yaml

```
text
name: Deploy CreditX (Prod)

on:
  push:
    branches: [ main ]
    paths:
      - "services/creditx-service/**"
      - "config/hyperlift/creditx-service.yaml"

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build image
        run: |
          cd services/creditx-service
          docker build -t spaceship.registry.io/ecosystem/creditx-
service:v2.0.0-dragonfly .

      - name: Push image
        run: |
          echo "$REGISTRY_PASSWORD" | docker login spaceship.registry.io
-u "$REGISTRY_USER" --password-stdin
          docker push spaceship.registry.io/ecosystem/creditx-
service:v2.0.0-dragonfly

      - name: Deploy via Hyperlift
        run: |
          spaceship hyperlift deploy \
            --config ../../config/hyperlift/creditx-service.yaml \
            --env production
        env:
          SPACESHIP_TOKEN: ${{ secrets.SPACESHIP_TOKEN }}
```

config/monitoring/prometheus-rules.yaml

```
text
groups:
  - name: dragonfly.rules
    rules:
      - alert: DragonflyHighLatency
        expr: dragonfly_request_latency_p95_ms > 10
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Dragonfly p95 latency > 10ms"
          description: "Cache latency elevated, investigate cache or
network."

      - alert: DragonflyLowHitRatio
        expr: dragonfly_cache_hit_ratio < 0.5
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Dragonfly cache hit ratio < 50%"
          description: "Cache misconfiguration or cold cache, check TTLs
and traffic."

      - alert: DragonflyDown
        expr: up{job="dragonfly"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Dragonfly cache is DOWN"
          description: "Failing over to DB only; investigate
immediately."
```

How to use this stack

- For infra:
    - Provision the Dragonfly VM + container according to `config/dragonfly-cluster.yaml`.
    - Ensure `dragonfly-cache.internal:6379` is reachable from all services.
- For services:
    - Wire Python services to `core_cache.py` and Node services to `core_cache.ts`.
    - Ensure env variables in `config/env.global.example.yaml` are provided via Spaceship/Hyperlift secrets.
- For agents:
    - Load `orchestrator_prompt.yaml`, `recovery_agent_p`

`rompt.yaml`, and `agent_registry.yaml` into your agent runtime / MCP layer.

- **For deployment:**
    - Apply Hyperlift manifests in `config/hyperlift/*.yaml`.
    - Enable the GitHub Actions workflows to get auto-deploys on `main`.

With these files in place, Phase 1 is a **Dragonfly-backed, Spaceship-native production build**: Redis is fully removed, Dragonfly is the standard cache layer, and the agents, services, and deployment controls all reflect that reality end-to-end