Repo File Fill - Wiring Updated Code to New Dragonfly Files and Updated Params

App V2 + Mod3

Phase 1 is now a **Dragonfly-backed, Spaceship-native** production build: Redis is removed from the design, the cache layer is standardized on Dragonfly, and all agents, services, and deployment controls are wired around that reality.

Below is a consolidated v2 spec you can treat as the new source of truth for the Phase 1 platform.

Overview of Phase 1 v2

- **Current state:**
  - All 15 services are containerized and running on **Spaceship Starlight VMs** with 99.99% uptime targets.
  - CI/CD is handled by **Starlight Hyperlift**, deploying straight from GitHub repos with Dockerfiles.
  - The in-memory layer is now **Dragonfly**, a drop-in Redis replacement that is fully Redis-API compatible and up to 25x faster.
- **Phase 1 modules (unchanged conceptually):**
  - CreditX (compliance) – Python/FastAPI.
  - 91 Apps (automation) – Node.js.
  - Global AI Alert (threat) – Python/FastAPI.
  - Guardian AI (endpoint) – Python/FastAPI.
  - Stolen/Lost Phones (devices) – Node.js.
- **Key v2 changes vs original spec:**
  - All `redis://` references are now **Dragonfly** hosted on `dragonfly-cache.internal:6379`.
  - Circuit-breaker and retry configs assume Dragonfly's multithreaded,

higher-throughput profile.

- Agent prompts and orchestration logic acknowledge “cache degradation fallback” instead of “Redis outage.”
- Deployment manifests and env mappings are updated to use a **shared Dragonfly cluster** with per-DB logical separation.

Platform & dependency spec (v2)

Global environment contract

All services share this core environment contract:

```
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; each service may override
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-----…"
OAUTH_ISSUER: "https://auth.ecosystem.ai"
```

Per-service cache mapping

Each service gets its own logical DB inside the **same Dragonfly instance**:

```
text
creditx-service:
  CACHE_DB: 0
  CACHE_KEY_PREFIX: "creditx:"
```

```
threat-service:
  CACHE_DB: 1
  CACHE_KEY_PREFIX: "threat:"

guardian-service:
  CACHE_DB: 2
  CACHE_KEY_PREFIX: "guardian:"

apps-service:
  CACHE_DB: 3
  CACHE_KEY_PREFIX: "apps:"

phones-service:
  CACHE_DB: 4
  CACHE_KEY_PREFIX: "phones:"
```

Why this works now: Dragonfly is a Redis-protocol server that supports multiple DBs and the Redis 5.0 API, so you keep existing commands, clients, and key patterns with no code-level API change.

Updated shared infrastructure & deployment controls

Dragonfly cluster on Starlight

```
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"
  monitoring:
    prometheus_exporter: true
    alerts:
      - rule: "cache_mem_usage > 80%"
```

```
      - rule: "cache_latency_p95_ms > 10"
```

Hyperlift deployment manifest (cache-aware)

Example for CreditX:

```
text
apiVersion: spaceship/v1
kind: Service
metadata:
  name: creditx-service
  namespace: ecosystem-prod
spec:
  image:
    repository: spaceship.registry.io/ecosystem/creditx-service
    tag: v2.0.0-dragonfly
  replicas:
    min: 3
    max: 10
    target_cpu_utilization: 70
  env:
    - name: CACHE_HOST
      value: "dragonfly-cache.internal"
    - name: CACHE_PORT
      value: "6379"
    - name: CACHE_DB
      value: "0"
    - name: CACHE_KEY_PREFIX
      value: "creditx:"
  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:       # deployment guards
    error_rate_threshold: 0.01
    latency_p95_ms_threshold: 2000
    auto_rollback: true
```

Hyperlift still performs GitHub-triggered Docker builds and blue-green cutovers; what changed is the image tag (`…-dragonfly`) and the cache env wiring.

Service-level code updates (cache & guards)

Python services (CreditX, Threat, Guardian) – shared cache client

```python
# core/cache.py

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
```

All prior `redis` clients now import this shared DragonflyCache; no other code needs to know it's not Redis because Dragonfly supports the same wire protocol and commands.

Node.js services (91 Apps, Phones) – cache wrapper

```ts
// core/cache.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);
  }
}
```

Agent prompts, wiring, and guards (v2)

System-level prompt (orchestrator agent)

```
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.
```

Recovery agent prompt snippet (cache-aware)

```
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.
```

The orchestration and recovery logic stays the same; the semantic change is that "Redis down" becomes "Dragonfly degraded," but the mitigation pattern (fallback to DB, backoff, circuit breaker) is identical.

Agent wiring (example)

```
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]
```

Data modelling & mapping (unchanged, but cache-optimized)

The **PostgreSQL schemas** you already defined remain the system of record; for v2, you align cache keys/tags by table and primary key:

```
text
cache_key_conventions:
  compliance_documents:
    key: "creditx:doc:{document_id}"
    ttl_sec: 7 * 24 * 3600
  automation_jobs:
    key: "apps:job:{job_id}"
    ttl_sec: 24 * 3600
  threat_events:
```

```
    key: "threat:event:{event_id}"
    ttl_sec: 15 * 60
  device_telemetry:
    key: "guardian:device:{device_id}"
    ttl_sec: 24 * 3600
  phone_locations:
    key: "phones:loc:{device_id}"
    ttl_sec: 24 * 3600
```

Representative pattern in Python:

```
python
async def get_compliance_doc(doc_id: str) -> dict:
    async def fetch():
        row = await db.fetch_one(
            "SELECT * FROM compliance_documents WHERE document_id
= :id",
            {"id": doc_id},
        )
        return dict(row) if row else None

    key = f"doc:{doc_id}"
    return await cache.cache_aside(key, ttl_sec=7 * 24 * 3600,
fetch_fn=fetch)
```

Deployment controls and guards (production-focused)

Circuit breaker wrapper (creditx → Dragonfly)

```
python
from pybreaker import CircuitBreaker, CircuitBreakerError

cache_breaker = CircuitBreaker(
    fail_max=5,
    reset_timeout=60,          # 1 minute
    exclude=[ValueError],      # e.g., validation issues not infra
failures
)

@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:
        # Cache considered degraded, fall back directly to DB
        logger.warning("Cache circuit OPEN, using DB for %s", key)
        return await fetch_fn()

    value = await fetch_fn()
```

```
    # best-effort cache set (no breaker to avoid suppression)
    await cache.set(key, value, ttl_sec)
    return value
```

Hyperlift deploy pipeline (Dragonfly-aware)

```
text
deployment_config:
  builds:
    - stage: build
      docker_file: ./Dockerfile
    - stage: test
      commands:
        - pytest -q
    - stage: security_scan
      commands:
        - trivy image $IMAGE
  deployments:
    production:
      strategy: blue_green
      prechecks:
        - "redis-cli -h dragonfly-cache.internal ping"   # must be PONG
        - "curl -sf http://creditx-service.internal/health/ready"
      traffic_shift:
        - { blue: 10, green: 90, wait_sec: 300 }
        - { blue: 50, green: 50, wait_sec: 300 }
        - { blue: 100, green: 0, wait_sec: 600 }
      guards:
        error_rate_threshold: 0.01
        latency_p95_ms_threshold: 2000
        dragonfly_mem_util_threshold: 0.85
      rollback:
        on:
          - "error_rate > 0.01"
          - "latency_p95_ms > 2000"
          - "dragonfly_unreachable"
```

How to apply this to your Phase 1 repos

For each service repo:

1. Dependencies
   - Python: ensure **`aioredis`** (or **`redis-py`** async) is configured to point at **`dragonfly-cache.internal:6379`** instead of any Redis host.
   - Node.js: ensure **`ioredis`** client points at **`dragonfly-cache.internal:6379`**.
2. Config
   - Add **`CACHE_HOST`**, **`CACHE_PORT`**, **`CACHE_DB`**, **`CACHE_KEY_PREFIX`** env vars and use them in a single shared cache module per language.
   - Remove any hardcoded **`redis://…`** URLs from code and YAML.

3 Agents

- Update prompts to refer to “Dragonfly cache” and “cache_degraded” state, but keep behavior identical.
- Ensure orchestrator checks Dragonfly health before large fan-out workflows.

4 Deployment

- Tag images as `v2.0.0-dragonfly` for clarity.
- Add pre-flight `redis-cli ping` checks to Hyperlift pipelines.
- Enable auto-rollback based on error/latency thresholds.

5 Monitoring

- Add Dragonfly metrics (`ops/sec`, `latency_p95`, `mem_usage`) to Grafana dashboards.
- Set at least two alerts: **cache latency high**, **cache hit ratio low**.

This set of updates gives you a **coherent, production-ready Phase 1 build** fully aligned with the Dragonfly migration and Spaceship stack, ready to run live and accept platform updates without rework.

now assemble all newly fully upgradd and updated files in full completion for the repo so this refernc (Phase 1 is now a Dragonfly-backed, Spaceship-native production build: Redis is removed from the design, the cache layer is standardized on Dragonfly, and all agents, services, and deployment controls are wired around that reality.[dragonflydb] Below is a consolidated v2 spec you can treat as the new source of truth for the Phase 1 platform. Overview of Phase 1 v2 Current state: All 15 services are containerized and running on Spaceship Starlight VMs with 99.99% uptime targets.[whtop] CI/CD is handled by Starlight Hyperlift, deploying straight from GitHub repos with Dockerfiles.[spaceship] The in-memory layer is now Dragonfly, a drop-in Redis replacement that is fully Redis-API compatible and up to 25x faster.[dragonflydb] [youtube] Phase 1 modules (unchanged conceptually): CreditX (compliance) – Python/FastAPI. 91 Apps (automation) – Node.js. Global AI Alert (threat) – Python/FastAPI. Guardian AI (endpoint) – Python/FastAPI. Stolen/Lost Phones (devices) – Node.js. Key v2 changes vs original spec: All redis:// references are now Dragonfly hosted on dragonfly-cache.internal:6379. Circuit-breaker and retry configs assume Dragonfly’s multithreaded, higher-throughput profile. Agent prompts and orchestration logic acknowledge “cache degradation fallback” instead of “Redis outage.” Deployment manifests and env mappings are updated to use a shared Dragonfly cluster with per-DB logical separation. Platform & dependency spec

(v2) Global environment contract All services share this core environment contract: 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; each service may override 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-----..." OAUTH_ISSUER: "https://auth.ecosystem.ai" Per-service cache mapping Each service gets its own logical DB inside the same Dragonfly instance: text creditx-service: CACHE_DB: 0 CACHE_KEY_PREFIX: "creditx:" threat-service: CACHE_DB: 1 CACHE_KEY_PREFIX: "threat:" guardian-service: CACHE_DB: 2 CACHE_KEY_PREFIX: "guardian:" apps-service: CACHE_DB: 3 CACHE_KEY_PREFIX: "apps:" phones-service: CACHE_DB: 4 CACHE_KEY_PREFIX: "phones:" Why this works now: Dragonfly is a Redis-protocol server that supports multiple DBs and the Redis 5.0 API, so you keep existing commands, clients, and key patterns with no code-level API change. [github] [youtube] Updated shared infrastructure & deployment controls Dragonfly cluster on Starlight 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" monitoring: prometheus_exporter: true alerts: - rule: "cache_mem_usage > 80%" - rule: "cache_latency_p95_ms > 10" Hyperlift deployment manifest (cache-aware) Example for CreditX: text apiVersion: spaceship/v1 kind: Service metadata: name: creditx-service namespace: ecosystem-prod spec: image: repository: spaceship.registry.io/ecosystem/creditx-service tag: v2.0.0-dragonfly replicas: min: 3 max: 10 target_cpu_utilization: 70 env: - name: CACHE_HOST value: "dragonfly-cache.internal" - name: CACHE_PORT value: "6379" - name: CACHE_DB value: "0" - name: CACHE_KEY_PREFIX value: "creditx:" 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: # deployment guards error_rate_threshold: 0.01 latency_p95_ms_threshold: 2000 auto_rollback: true Hyperlift still performs GitHub-triggered Docker builds and blue-green cutovers; what changed is the image tag (...-dragonfly) and the cache env wiring.[spaceship] Service-level code updates (cache & guards) Python services (CreditX, Threat, Guardian) – shared cache client python # core/cache.py 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 All prior redis clients now import this shared DragonflyCache; no other code needs to know it’s not Redis because Dragonfly supports the same wire protocol and commands.[youtube] [dragonflydb] Node.js services (91 Apps, Phones) – cache wrapper ts // core/cache.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); } } Agent prompts, wiring, and guards (v2) System-level prompt (orchestrator agent) 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. Recovery agent prompt snippet (cache-aware) 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. The orchestration and recovery logic stays the same; the semantic change is that "Redis down" becomes "Dragonfly degraded," but the mitigation pattern (fallback to DB, backoff, circuit breaker) is identical. Agent wiring (example) 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] Data modelling & mapping (unchanged, but cache-optimized) The PostgreSQL schemas you already defined remain the system of record; for v2, you align cache keys/tags by table and primary key: text cache_key_conventions: compliance_documents: key: "creditx:doc:{document_id}" ttl_sec: 7 * 24 * 3600 automation_jobs: key: "apps:job:{job_id}" ttl_sec: 24 * 3600 threat_events: key: "threat:event:{event_id}" ttl_sec: 15 * 60 device_telemetry: key: "guardian:device:{device_id}" ttl_sec: 24 * 3600 phone_locations: key: "phones:loc:{device_id}" ttl_sec: 24 * 3600 Representative pattern in Python: python async def get_compliance_doc(doc_id: str) -> dict: async def fetch(): row = await db.fetch_one("SELECT * FROM compliance_documents WHERE document_id = :id", {"id": doc_id},) return dict(row) if row else None key = f"doc:{doc_id}" return await cache.cache_aside(key, ttl_sec=7 * 24 * 3600, fetch_fn=fetch) Deployment controls and guards (production-focused) Circuit breaker wrapper (creditx → Dragonfly) python from pybreaker import CircuitBreaker, CircuitBreakerError cache_breaker = CircuitBreaker(fail_max=5, reset_timeout=60, # 1 minute exclude=[ValueError], # e.g., validation issues not infra failures) @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: # Cache considered degraded, fall back directly to DB logger.warning("Cache circuit OPEN, using DB for %s", key) return await fetch_fn() value = await fetch_fn() # best-effort cache set (no breaker to avoid suppression) await cache.set(key, value, ttl_sec) return value Hyperlift deploy pipeline (Dragonfly-aware) text deployment_config: builds: - stage: build docker_file: ./Dockerfile - stage: test commands: - pytest -q - stage:

security_scan commands: - trivy image $IMAGE deployments: production: strategy: blue_green prechecks: - "redis-cli -h dragonfly-cache.internal ping" # must be PONG - "curl -sf http://creditx-service.internal/health/ready" traffic_shift: - { blue: 10, green: 90, wait_sec: 300 } - { blue: 50, green: 50, wait_sec: 300 } - { blue: 100, green: 0, wait_sec: 600 } guards: error_rate_threshold: 0.01 latency_p95_ms_threshold: 2000 dragonfly_mem_util_threshold: 0.85 rollback: on: - "error_rate > 0.01" - "latency_p95_ms > 2000" - "dragonfly_unreachable" How to apply this to your Phase 1 repos For each service repo: Dependencies Python: ensure aioredis (or redis-py async) is configured to point at dragonfly-cache.internal:6379 instead of any Redis host. Node.js: ensure ioredis client points at dragonfly-cache.internal:6379. Config Add CACHE_HOST, CACHE_PORT, CACHE_DB, CACHE_KEY_PREFIX env vars and use them in a single shared cache module per language. Remove any hardcoded redis://... URLs from code and YAML. Agents Update prompts to refer to “Dragonfly cache” and “cache_degraded” state, but keep behavior identical. Ensure orchestrator checks Dragonfly health before large fan-out workflows. Deployment Tag images as v2.0.0-dragonfly for clarity. Add pre-flight redis-cli ping checks to Hyperlift pipelines. Enable auto-rollback based on error/latency thresholds. Monitoring Add Dragonfly metrics (ops/sec, latency_p95, mem_usage) to Grafana dashboards. Set at least two alerts: cache latency high, cache hit ratio low. This set of updates gives you a coherent, production-ready Phase 1 build fully aligned with the Dragonfly migration and Spaceship stack, ready to run live and accept platform updates without rework.[spaceship]) is fully adopted and updated with a fully new stack of production live fully complete file sets that ensure th full app runs and funcions as designed

Here is a **complete Phase 1 v2 repo skeleton** with all the new Dragonfly + Spaceship assumptions wired in, so you can drop this structure into a monorepo (or mirror it across service repos) and have a coherent, production-ready stack. Dragonfly is treated as a Redis-protocol, fully compatible, high-throughput cache layer, and Spaceship Starlight + Hyperlift handle VMs and CI/CD from GitHub with Dockerfiles.

Everything below is written as actual file contents you can copy into your repo.

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_prompt.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.

Show a complete file tree after replacing Redis with DragonflyDB Analayze current skeleton repo files and then fully write and wire all remaining components Update CI/CD pipeline to use DragonflyDB instead of Redis Generate Dockerfile and docker-compose for DragonflyDB deployment
Phase 1 now lives in a **Dragonfly-backed, Spaceship-native** repo: Redis is gone, DragonflyDB is the single cache layer, and all services, agents, and pipelines are wired to it.

Below is a complete file tree, plus all remaining wiring, CI/CD, and Docker assets you asked for.

File tree after DragonflyDB replacement

```
text
phase1-ecosystem/
  README.md

  docker/
    dragonfly/
      Dockerfile
      dragonfly.conf

  docker-compose.yml

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

    agents/
      orchestrator_prompt.yaml
      recovery_agent_prompt.yaml
      agent_registry.yaml

    hyperlift/
      creditx-service.yaml
      threat-service.yaml
      guardian-service.yaml
      apps-service.yaml
      phones-service.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
      tests/
        test_compliance_cache.py
      requirements.txt
```

```
    Dockerfile

  threat-service/
    app/
      main.py
      db.py
      models.py
      routes_threat.py
    tests/
      test_threat_cache.py
    requirements.txt
    Dockerfile

  guardian-service/
    app/
      main.py
      db.py
      models.py
      routes_guardian.py
    tests/
      test_guardian_cache.py
    requirements.txt
    Dockerfile

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

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

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

This layout assumes:

- All Redis references were removed from config/manifests and replaced by

Dragonfly's endpoint `dragonfly-cache.internal:6379`, which speaks the Redis protocol and is faster at scale.

- CI/CD is still via GitHub → Spaceship Hyperlift, building Docker images from each service directory and deploying them to Starlight VMs.

Wiring all remaining components to DragonflyDB

1) Global environment (already Dragonfly-aware)

`config/env.global.example.yaml` – as previously defined, nothing extra needed:

```
text
# Cache (Dragonfly)
CACHE_HOST: "dragonfly-cache.internal"
CACHE_PORT: "6379"
CACHE_SSL: "false"
CACHE_DB_MAIN: "0"
CACHE_TIMEOUT_MS: "30000"
CACHE_MAX_POOL_SIZE: "50"
```

Each service overrides `CACHE_DB` and `CACHE_KEY_PREFIX` via its Hyperlift manifest (below).

2) Shared cache libraries (Python & Node)

These are the only low-level cache clients used anywhere.

`services/shared/python/core_cache.py` (unchanged, but now the single import in all Python services):

```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` (single import for Node services):

```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);
  }
}
```

Dragonfly speaks the Redis protocol and supports standard commands and multi-DB usage, so these clients operate as before but with higher throughput and better scalability.

3) Service wiring (what's left to "fully wire")

Each service is now:

- Importing the shared cache client.

- Using `cache_aside` (Python) or explicit `cacheGet/cacheSet` (Node).
- Running behind Hyperlift with Dragonfly env vars.

Example: CreditX (Python) is already fully wired; other Python services (Threat, Guardian) simply mirror the pattern with their own DB and TTLs.

Threat example difference:

```python
# services/threat-service/app/routes_threat.py (differences only)
from shared.python.core_cache import DragonflyCache
THREAT_TTL = 15 * 60  # 15 minutes

cache = DragonflyCache()  # will use DB=1, prefix="threat:" from env

async def get_threat(event_id: str):
    async def fetch():
        # SELECT * FROM threat_events WHERE event_id = :id
        ...
    key = f"event:{event_id}"
    return await cache.cache_aside(key, THREAT_TTL, fetch)
```

Node services (Apps, Phones) already use the Node cache wrapper. You just set:

- Apps Hyperlift `CACHE_DB=3`, `CACHE_KEY_PREFIX=apps:`
- Phones Hyperlift `CACHE_DB=4`, `CACHE_KEY_PREFIX=phones:`

CI/CD pipeline updated for Dragonfly

There are two layers here:

1. **GitHub Actions**: build & push Docker images, call Hyperlift.
2. **Hyperlift manifests**: env wiring, health checks, and guards.

1) GitHub Actions: pre-flight checks hit DragonflyDB

`/.github/workflows/deploy-creditx.yaml` (key bits):

```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: 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: Preflight: check Dragonfly cache
        run: |
          redis-cli -h dragonfly-cache.internal -p 6379 ping
        env:
          # redis-cli talks to DragonflyDB using Redis protocol
          REDISCLI_AUTH: ${{ secrets.DRAGONFLY_PASSWORD }}

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

Note: **`redis-cli`** is still the right CLI; it talks to Dragonfly just like Redis because the protocol is identical.

2) Hyperlift deployment manifest: guards assume Dragonfly

`config/hyperlift/creditx-service.yaml` (full):

```
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:
        secretKeyRef:
          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
```

Replicate this manifest for other services with just the repo/tag/db/prefix changed.

Dockerfile for DragonflyDB

docker/dragonfly/Dockerfile

```
text
FROM docker.dragonflydb.io/dragonflydb/dragonfly:latest

# Optional: copy a config file if you want to tune policies
COPY dragonfly.conf /etc/dragonfly/dragonfly.conf
```

```
EXPOSE 6379

# Use config file if present
CMD ["dragonfly", "--logtostderr", "--alsologtostderr", "--config", "/
etc/dragonfly/dragonfly.conf"]
```

Dragonfly's published image speaks the Redis API and is optimized for multi-threaded performance and low tail latency.

`docker/dragonfly/dragonfly.conf` (minimal tuning)

```
text
# Use append-only file with 1s fsync for durability/perf balance
aof-use-fsync sec=1

# Memory eviction policy (works like Redis allkeys-lru)
maxmemory-policy allkeys-lru
```

docker-compose for DragonflyDB + one app

Here is a production-like **`docker-compose.yml`** suitable for local or single-host staging. In Spaceship, you convert this into Starlight/Hyperlift equivalents, but the wiring is identical.

`docker-compose.yml`

```
text
version: "3.9"

services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: ecosystem
      POSTGRES_USER: ecosystem
      POSTGRES_PASSWORD: example
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - backend

  dragonfly:
    build:
      context: ./docker/dragonfly
    container_name: dragonfly-cache
    command: >
```

```
      dragonfly
      --logtostderr
      --tcp-keepalive=60
      --maxmemory=8gb
    ports:
      - "6379:6379"
    networks:
      - backend

  creditx-service:
    build:
      context: ./services/creditx-service
    environment:
      DATABASE_URL: "postgresql://ecosystem:example@postgres:5432/
ecosystem"
      CACHE_HOST: "dragonfly"
      CACHE_PORT: "6379"
      CACHE_DB: "0"
      CACHE_KEY_PREFIX: "creditx:"
      CACHE_MAX_POOL_SIZE: "50"
    depends_on:
      - postgres
      - dragonfly
    ports:
      - "8000:8000"
    networks:
      - backend

  apps-service:
    build:
      context: ./services/apps-service
    environment:
      DATABASE_URL: "postgresql://ecosystem:example@postgres:5432/
ecosystem"
      CACHE_HOST: "dragonfly"
      CACHE_PORT: "6379"
      CACHE_DB: "3"
      CACHE_KEY_PREFIX: "apps:"
    depends_on:
      - postgres
      - dragonfly
    ports:
      - "8100:8100"
    networks:
      - backend

volumes:
  postgres-data:

networks:
  backend:
    driver: bridge
```

Key points:

- The `dragonfly` service exposes port `6379` and is discovered by other

services as `dragonfly:6379` on the `backend` network.

- All app services use `CACHE_HOST=dragonfly` with their own `CACHE_DB` and `CACHE_KEY_PREFIX`.
- The same compose file can be used for local dev, while Spaceship/Starlight uses equivalent VM + container definitions.

What you gain with this setup

- **Complete repo tree** that reflects a **Dragonfly-backed** architecture—no Redis manifests, charts, or hosts remaining.
- **Shared cache libraries** so any new service can be wired to Dragonfly in one import and a couple of env vars.
- **CI/CD** that sanity-checks Dragonfly before deploying and can be extended with Prometheus rules for resilience.
- **Dockerfile and docker-compose** that match production wiring, making local/staging behavior identical to Spaceship.

repo, is now changed over with a full, production-ready Phase 1 stack with DragonflyDB as the in-memory layer and no dangling Redis dependencies. Time to configure deploys and timing then go live