codebase repo file updates:

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 mod 3

- **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:

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.[spaceship]