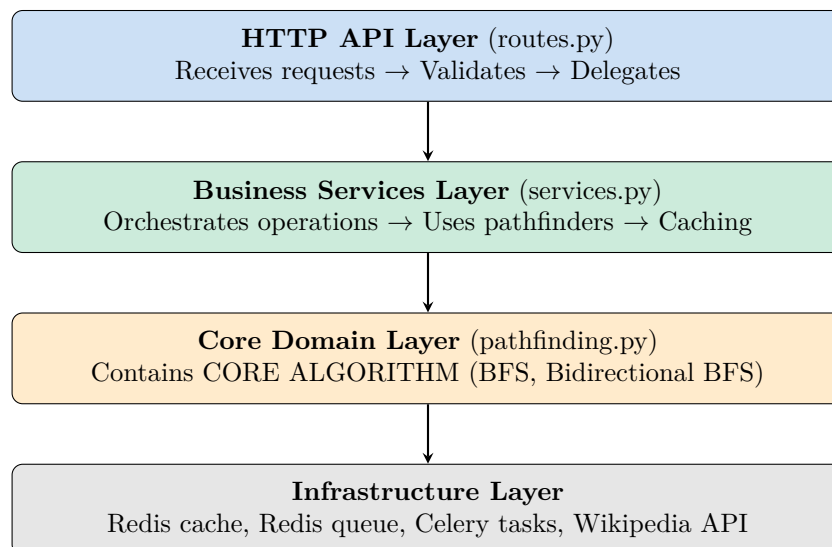


Iris Backend Architecture Guide

Wikipedia Pathfinding Engine



Interview Preparation & Development Reference

Last Updated: January 2026

Contents

1	Project Structure Overview	2
2	Architecture Layers	2
2.1	Layer Descriptions	2
2.2	Data Flow Diagram	3
3	Architecture Decisions & Engineering Rationale	3
3.1	High-Level Architecture Decisions	3
3.2	Low-Level Implementation Decisions	4
3.3	Scalability Characteristics Summary	5
4	Service Lifecycle & Initialization	5
4.1	The “When”: Lazy Initialization	5
4.2	Component Lifecycle Table	6
4.3	Initialization Flow	6
5	Progress Callback System (Real-Time Updates)	6
5.1	The Problem	6
5.2	The Solution: Callback Injection	6
5.3	Code Flow	6
5.3.1	Step 1: Task Defines the Callback	6
5.3.2	Step 2: Factory Passes Callback to PathFinder	7
5.3.3	Step 3: BFS Calls the Callback Periodically	7
5.3.4	Step 4: Client Sees Progress via Polling	8
5.4	Architecture Diagram: Callback Flow	8
5.5	Why This Design?	8
6	Dependency Injection (DI) Explained	8
6.1	The Pattern: Singleton + Factory	9
6.2	Swapping Implementations	9
7	File-by-File Breakdown	9
7.1	Core Files	9
7.2	Infrastructure Files	10
8	Caching Strategy	10
9	Interview Q&A Cheat Sheet	10
10	Quick Reference: Where to Make Changes	12
11	API Endpoints Reference	12
12	Configuration Reference	12

1 Project Structure Overview

The Iris backend follows a **Clean Architecture** pattern with clear separation of concerns.

```

1 iris-web-backend/
2 |-- app/                                # Main application package
3 |   |-- __init__.py                    # Flask app factory & Celery setup
4 |   |-- api/                          # HTTP API layer
5 |   |   |-- routes.py                 # API endpoints (Flask blueprints)
6 |   |   |-- middleware.py             # Decorators: error handling, logging, CORS
7 |   |   |-- schemas.py               # Marshmallow schemas for validation
8 |   |-- core/                         # Core business logic (THE BRAIN)
9 |   |   |-- factory.py               # Dependency Injection (ServiceFactory)
10 |   |   |-- interfaces.py            # Abstract interfaces (contracts)
11 |   |   |-- models.py               # Data models (dataclasses)
12 |   |   |-- pathfinding.py          # CORE ALGORITHM (BFS)
13 |   |   |-- services.py             # Business services orchestration
14 |   |-- external/                   # External API integrations
15 |   |   |-- wikipedia.py            # Wikipedia API client
16 |   |-- infrastructure/             # Infrastructure concerns
17 |   |   |-- cache.py                # Redis cache implementation
18 |   |   |-- redis_queue.py          # Redis queue for BFS state
19 |   |   |-- tasks.py                # Celery background tasks
20 |   |-- utils/                      # Utilities
21 |       |-- exceptions.py           # Custom exception hierarchy
22 |       |-- logging.py              # Logging configuration
23 |-- config/                          # Configuration classes
24 |-- tests/                           # Test suites
25 |-- run.py                           # Application entry point
26 |-- celery_worker.py                 # Celery worker entry point

```

Listing 1: Project Directory Structure

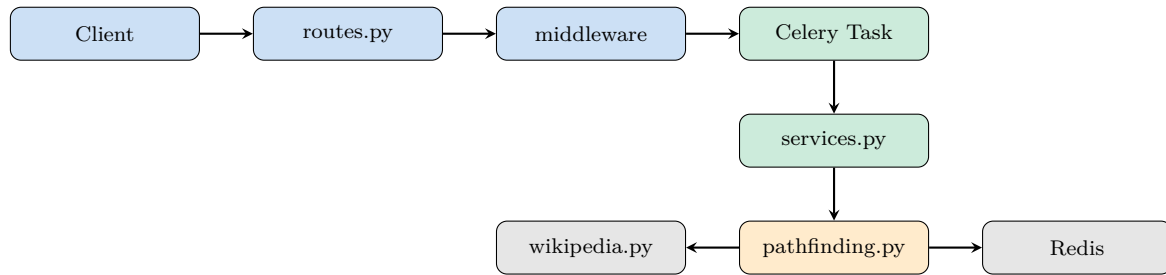
2 Architecture Layers

The codebase follows a **layered architecture** similar to Clean Architecture. Each layer has a single responsibility.

2.1 Layer Descriptions

Layer	Responsibility
HTTP API Layer <code>app/api/</code>	Receives HTTP requests, validates input using Marshmallow schemas, applies cross-cutting middleware (CORS, logging, error handling), delegates to business services, and formats JSON responses.
Business Services <code>app/core/services.py</code>	Orchestrates business operations. Checks cache before invoking pathfinding, coordinates between pathfinders and Wikipedia client, encapsulates use cases like “find path” or “explore page.”
Core Domain <code>app/core/pathfinding.py</code>	Contains the pure BFS algorithm. No knowledge of HTTP, Redis, or Flask. Only knows about abstract interfaces. This is where the graph traversal logic lives.
Infrastructure <code>app/infrastructure/</code>	Concrete implementations: Redis cache, Redis queue for BFS state, Celery tasks for background processing, Wikipedia API client. All “how” details live here.

2.2 Data Flow Diagram



3 Architecture Decisions & Engineering Rationale

This section justifies key technical decisions from a **distributed systems** and **scalability** perspective. These are the answers that demonstrate engineering depth.

3.1 High-Level Architecture Decisions

Decision: Asynchronous Task Processing with Celery

Problem: Wikipedia pathfinding can take 30 seconds to 5+ minutes. HTTP requests would timeout.

Decision: Offload pathfinding to Celery background workers. API returns a task ID immediately; client polls for results.

Why This Scales:

- **Horizontal Scaling:** Spin up N Celery workers to handle N concurrent searches.
- **Decoupling:** Web servers handle HTTP; workers handle compute. Each scales independently.
- **Resilience:** If a worker crashes, the task stays in Redis and another worker picks it up.

Trade-off: Added complexity of polling. Considered WebSockets but polling is simpler to implement and debug.

Decision: Redis as Cache

Problem: Need caching, a job queue, and result storage. Using 3 different systems increases operational overhead.

Decision: Use Redis for all three: Wikipedia link cache, Celery broker, and Celery result backend.

Why This Scales:

- **Single Point of Expertise:** Operations team only needs to master one system.
- **Redis Cluster:** Can scale to hundreds of thousands of operations/second.
- **Memory Efficiency:** Redis is optimized for this exact use case.

Trade-off: Redis is a single point of failure. Mitigated with Redis Sentinel or Redis Cluster in production.

Decision: Stateless Web Servers

Problem: Need to scale web tier horizontally behind a load balancer.

Decision: All application state lives in Redis. Flask servers are completely stateless.

Why This Scales:

- **Instant Horizontal Scaling:** Add/remove web servers without session migration.
- **Zero Downtime Deploys:** Rolling restarts don't lose user state.
- **Container-Friendly:** Perfect for Kubernetes/Docker deployments.

Decision: Separation of API and Worker Deployments

Problem: API servers need to respond quickly; workers need CPU for graph traversal.

Decision: Deploy API servers and Celery workers as separate services with different resource profiles.

Why This Scales:

- **Independent Scaling:** Scale workers based on queue depth; scale API based on request rate.
- **Resource Isolation:** Workers can use 100% CPU without affecting API latency.
- **Cost Optimization:** Use different instance types (API: network-optimized; Workers: compute-optimized).

3.2 Low-Level Implementation Decisions**Decision: Redis-Based BFS Queue Instead of In-Memory Deque**

Problem: BFS at depth 5+ can involve millions of nodes. Python's `deque` would exhaust worker memory.

Decision: Store BFS queue and visited set in Redis with session-specific keys.

Engineering Justification:

- **Memory Safety:** Worker memory stays constant regardless of search depth.
- **Crash Recovery:** If worker dies, search state persists (though we restart fresh currently).
- **Observability:** Can inspect queue depth via Redis CLI for debugging.

Trade-off: Higher latency per operation (network hop to Redis). Mitigated by batching operations.

Decision: Connection Pooling via Singleton Pattern

Problem: Creating new TCP connections to Redis/Wikipedia on every request is expensive.

Decision: Use lazy-loaded singletons for `RedisClient` and `WikipediaClient`.

Engineering Justification:

- **Redis:** Uses `redis.ConnectionPool` – amortizes TCP handshake across requests.
- **Wikipedia:** Uses `requests.Session` – enables HTTP Keep-Alive and connection reuse.
- **TLS Overhead:** Wikipedia uses HTTPS. Reusing connections avoids repeated TLS handshakes (100-300ms saved per request).

Decision: Bulk Wikipedia API Requests with `ThreadPoolExecutor`

Problem: Fetching links for 50 pages sequentially takes 50x the latency of one request.

Decision: Batch pages into groups of 50 (Wikipedia API limit) and fetch in parallel using `ThreadPoolExecutor`.

Engineering Justification:

- **I/O Bound:** Wikipedia API calls are network-bound; threads maximize throughput.
- **Configurable:** `WIKIPEDIA_MAX_WORKERS` controls parallelism (default: 10).
- **Rate Limiting:** Implicit rate limiting via thread pool size prevents hammering Wikipedia.

Decision: Cache TTLs Tuned to Data Volatility

Problem: Wikipedia pages change, but not frequently. Need to balance freshness vs. cache hit rate.

Decision: Different TTLs for different data types.

Engineering Justification:

- `wiki_links:*` – 24 hours. Page links rarely change; high reuse value.
- `path:*:*` – 1 hour. Paths are computed results; shorter TTL ensures freshness.
- `bfs_*` – 1 hour with explicit cleanup. Ephemeral search state; cleaned after use.

Decision: UUID-Based Session Isolation for BFS State

Problem: Multiple concurrent searches must not interfere with each other's BFS state.

Decision: Each search generates a UUID. All Redis keys are prefixed: `bfs_queue:<uuid>`, `bfs_visited:<uuid>:*`.

Engineering Justification:

- **No Locking Required:** Each search operates on its own keypace.
- **Easy Cleanup:** Pattern-based deletion: `DEL bfs_*:<uuid>:*`.
- **Debugging:** Can trace a specific search by its UUID in logs and Redis.

Decision: Dependency Injection via Factory Pattern

Problem: Need testable code without complex DI frameworks.

Decision: Hand-rolled `ServiceFactory` with constructor injection.

Engineering Justification:

- **Testability:** Tests inject mocks directly via constructor.
- **No Magic:** Unlike `@inject` decorators, the wiring is explicit and traceable.
- **Cleanup Method:** `ServiceFactory.cleanup()` resets singletons between tests.

3.3 Scalability Characteristics Summary

Dimension	How It Scales
Concurrent Searches	Add more Celery workers (horizontal).
API Request Rate	Add more Flask/Gunicorn instances behind load balancer.
Cache Size	Redis Cluster with sharding or increase instance memory.
Wikipedia Rate Limits	Configurable thread pool; add delay; use multiple API keys.
Storage (Results)	Redis with TTL-based eviction; results auto-expire after 1 hour.

4 Service Lifecycle & Initialization

Understanding precisely **when**, **where**, and **how** services are initialized demonstrates production awareness.

4.1 The “When”: Lazy Initialization

Services are **not** initialized when you run `python run.py`. They are created **on first use**.

- **Flask Process:** First API request triggers service initialization.
- **Celery Worker:** First task pickup triggers service initialization.

4.2 Component Lifecycle Table

Component	Lifecycle	Reasoning
Redis Client	Singleton	Maintains <code>ConnectionPool</code> ; avoids TCP handshake overhead per request.
Wikipedia Client	Singleton	Holds <code>requests.Session</code> ; enables HTTP Keep-Alive and TLS session reuse.
Cache Service	Singleton	Shared serialization logic wrapping the Redis client.
Pathfinding Service	Transient	New instance per task. Allows request-specific <code>progress_callback</code> and algorithm.

4.3 Initialization Flow

When a Celery task starts:

1. `find_path_task` starts in `infrastructure/tasks.py`.
2. Calls `get_pathfinding_service(algorithm, progress_callback)`.
3. `ServiceFactory` checks `_wikipedia_client`. It's `None` → creates it.
4. To create Wikipedia client, needs `_cache_service`. It's `None` → creates it.
5. To create cache, needs `_redis_client`. It's `None` → creates it.
6. Dependencies wired together via constructor injection; `PathFindingService` returned.
7. **Next task:** Singletons already exist → instant return.

Important

Singletons are Per-Process. If Gunicorn runs 4 workers, there are 4 independent Wikipedia Clients. They share state via the common Redis database, not via shared memory.

5 Progress Callback System (Real-Time Updates)

The progress callback allows clients to see **live search progress** rather than waiting blindly.

5.1 The Problem

A BFS search can take minutes. Without progress updates, the client has no idea:

- Is the search still running or stuck?
- How deep has it gone?
- How many nodes has it explored?

5.2 The Solution: Callback Injection

1. **Task creates a callback function** that updates Celery task state.
2. **Callback is injected** into `PathFindingService` via constructor.
3. **BFS algorithm calls the callback** every N nodes explored.
4. **Client polls** `/tasks/status/<id>` and sees real-time progress.

5.3 Code Flow

5.3.1 Step 1: Task Defines the Callback

```

1 @celery.task(bind=True)
2 def find_path_task(self, start_page, end_page, algorithm="bfs"):
3
4     # Define a closure that captures 'self' (the Celery task)
5     def progress_update(progress_data):
6         self.update_state(
7             state="PROGRESS",
8             meta=progress_data # Contains nodes_explored, current_depth, etc.
9         )
10
11    # Pass the callback to the service factory
12    pathfinding_service = get_pathfinding_service(
13        algorithm=algorithm,
14        progress_callback=progress_update # <-- Injected here
15    )
16
17    result = pathfinding_service.find_path(search_request)

```

Listing 2: tasks.py – Creating the Progress Callback

5.3.2 Step 2: Factory Passes Callback to Pathfinder

```

1 @classmethod
2 def create_pathfinding_service(cls, algorithm="bfs", progress_callback=None):
3     # ... get singletons ...
4
5     path_finder = RedisBasedBFSPathFinder(
6         wikipedia_client,
7         cache_service,
8         queue_service,
9         max_depth,
10        batch_size,
11        progress_callback # <-- Passed to the algorithm
12    )
13
14    return PathFindingService(path_finder, cache_service, wikipedia_client)

```

Listing 3: factory.py – Injecting the Callback

5.3.3 Step 3: BFS Calls the Callback Periodically

```

1 def _perform_bfs_search(self, start_page, end_page, ...):
2     nodes_explored = 0
3
4     while self.queue_service.length(queue_key) > 0:
5         current_item = self.queue_service.pop(queue_key)
6         nodes_explored += 1
7
8         # Report progress every 3 nodes
9         if self.progress_callback and nodes_explored % 3 == 0:
10             self.progress_callback({
11                 "status": "Searching...",
12                 "search_stats": {
13                     "nodes_explored": nodes_explored,
14                     "current_depth": current_depth,
15                     "last_node": current_page,
16                     "queue_size": self.queue_service.length(queue_key),
17                 },
18                 "search_time_elapsed": time.time() - start_time,
19             })
20
21     # ... rest of BFS logic ...

```


Listing 4: pathfinding.py – Invoking the Callback

5.3.4 Step 4: Client Sees Progress via Polling

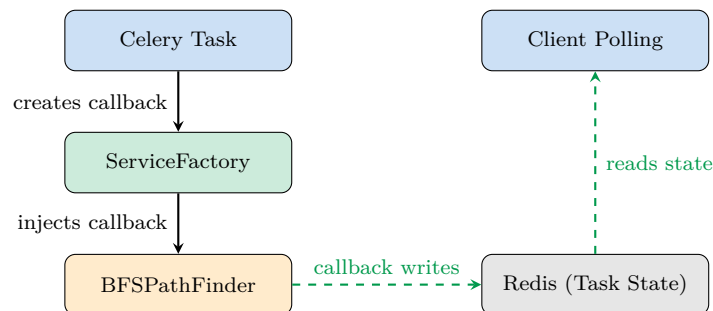
```

1 GET /tasks/status/abc-123
2
3 {
4     "status": "IN_PROGRESS",
5     "task_id": "abc-123",
6     "progress": {
7         "status": "Searching...",
8         "search_stats": {
9             "nodes_explored": 42,
10            "current_depth": 3,
11            "last_node": "Machine learning",
12            "queue_size": 150
13        },
14        "search_time_elapsed": 5.2
15    }
16 }

```

Listing 5: Client Polling Response

5.4 Architecture Diagram: Callback Flow



5.5 Why This Design?

Benefit	Explanation
Decoupling	The BFS algorithm doesn't know about Celery. It just calls a function.
Testability	In tests, pass a mock callback that records calls instead of updating Celery.
Flexibility	Easy to change reporting frequency or add more metrics without touching BFS.
No Polling Inside Worker	Worker doesn't poll; it pushes. Client pulls on its own schedule.

6 Dependency Injection (DI) Explained

Note

Location: app/core/factory.py – ServiceFactory class

6.1 The Pattern: Singleton + Factory

```

1 class ServiceFactory:
2     _redis_client = None          # Singleton storage
3     _cache_service = None
4     _wikipedia_client = None
5
6     @classmethod
7     def get_cache_service(cls):
8         if cls._cache_service is None:
9             redis_client = cls.get_redis_client()
10            cls._cache_service = RedisCache(redis_client)
11            return cls._cache_service
12
13    @classmethod
14    def create_pathfinding_service(cls, algorithm="bfs", progress_callback=None):
15        # Retrieve singletons, then inject into new instance
16        wiki = cls.get_wikipedia_client()
17        cache = cls.get_cache_service()
18        queue = cls.get_queue_service()
19
20        path_finder = RedisBasedBFSPathFinder(wiki, cache, queue,
21        progress_callback)
22        return PathFindingService(path_finder, cache, wiki)

```

Listing 6: ServiceFactory Implementation

6.2 Swapping Implementations

To use Memcached instead of Redis:

1. Create MemcachedCache implementing CacheServiceInterface.
2. Modify ServiceFactory.get_cache_service() to return MemcachedCache.
3. Done. All services automatically use the new cache.

7 File-by-File Breakdown

7.1 Core Files

File	Purpose
app/core/pathfinding.py	CORE ALGORITHM. Redis-based BFS. Modify for algorithm changes.
app/core/services.py	Business orchestration. Cache checking, validation, timing.
app/core/factory.py	DI container. Manages singletons and object creation.
app/core/interfaces.py	Abstract contracts for testability and loose coupling.
app/core/models.py	Dataclasses for requests, results, and DTOs.

7.2 Infrastructure Files

File	Purpose
app/infrastructure/cache.py	Redis cache implementation with connection pooling.
app/infrastructure/redis_queue.py	FIFO queue for BFS state (push, pop, length).
app/infrastructure/tasks.py	Celery task definitions with retry logic and progress updates.
app/external/wikipedia.py	Wikipedia API client with bulk fetching and redirect handling.

8 Caching Strategy

Key Pattern	TTL	Purpose
wiki_links:{page}	24h	Caches Wikipedia page links. High reuse.
path:{start}:{end}	1h	Caches completed path results.
bfs_visited:<uuid>*	1h	Temporary visited set. Cleaned after search.
bfs_paths:<uuid>*	1h	Temporary path tracking. Cleaned after search.
bfs_queue:<uuid>	Session	BFS queue. Explicitly deleted after search.

9 Interview Q&A Cheat Sheet

Q: If you had to modify the core pathfinding algorithm

Answer: app/core/pathfinding.py – specifically the `RedisBasedBFSPathFinder` class and its `find_shortest_path()` method.

Q: When and where do services get initialized?

Answer: Services are **lazily initialized** by `ServiceFactory` (app/core/factory.py) on first use. Not at app startup. Flask: first request. Celery: first task.

Q: Which objects are reused vs. created fresh?

Answer:

- **Reused (Singletons):** Redis Client, Wikipedia Client, Cache Service. Maintains connection pools.
- **Fresh (Transient):** PathFindingService. New per task for isolated callbacks.

Q: How does the progress callback work?

Answer:

1. Celery task defines a closure that calls `self.update_state()`.
2. This closure is passed to `ServiceFactory.create_pathfinding_service()`.
3. Factory injects it into `RedisBasedBFSPathFinder` constructor.
4. BFS calls the callback every N nodes, writing progress to Redis.
5. Client polls `/tasks/status/<id>` to read progress.

Q: Why use Celery instead of handling pathfinding synchronously?

Answer: Pathfinding can take minutes. Synchronous handling would:

- Timeout HTTP connections (typically 30-60s limit).
- Block web server threads, reducing throughput.
- Provide no progress visibility to users.

Celery enables horizontal scaling, resilience, and real-time progress.

Q: Why Redis for BFS queue instead of Python deque?

Answer: Memory safety. BFS at depth 5+ can involve millions of nodes. Redis:

- Offloads memory from worker process.
- Enables inspection via Redis CLI.
- Provides crash-recovery potential (state persists).

Trade-off: Network latency per operation (mitigated by batching).

Q: How does the system scale horizontally?

Answer:

- **API:** Stateless Flask servers behind load balancer.
- **Workers:** Add Celery workers to increase search throughput.
- **Cache:** Redis Cluster for sharding and replication.

Q: Explain singletons across processes.

Answer: Singletons are per-process. 4 Gunicorn workers = 4 Wikipedia Clients. They share state via Redis (the database is the shared memory), not in-process memory.

Q: Why is the Wikipedia Client a singleton?

Answer: It holds a `requests.Session` which provides:

- HTTP Keep-Alive (connection reuse).
- TLS session resumption (avoids 100-300ms handshake).
- Cookie persistence if needed.

Creating new clients per request would destroy these benefits.

Q: How would you add a new pathfinding algorithm?

Answer:

1. Create class implementing `PathFinderInterface` in `pathfinding.py`.
2. Add case in `ServiceFactory.create_pathfinding_service()`.
3. Update API schema to accept new algorithm name.
4. Add tests in `tests/integration/test_pathfinding.py`.

10 Quick Reference: Where to Make Changes

Goal	Location
Modify BFS algorithm	app/core/pathfinding.py
Add new API endpoint	app/api/routes.py
Change request validation	app/api/schemas.py
Add new Celery task	app/infrastructure/tasks.py
Swap cache implementation	Implement interface, update factory.py
Add new exception type	app/utils/exceptions.py
Change configuration	config/base.py
Modify Wikipedia API logic	app/external/wikipedia.py

11 API Endpoints Reference

Method	Endpoint	Description
POST	/getPath	Start pathfinding. Returns task ID.
GET	/tasks/status/<id>	Poll for task status and progress.
POST	/explore	Explore links from a page.
GET	/health	System health check.
GET	/api	API information.
POST	/cache/clear	Admin: clear cache by pattern.

12 Configuration Reference

Variable	Default	Description
REDIS_URL	localhost:6379	Redis connection URL.
MAX_SEARCH_DEPTH	6	Maximum BFS depth.
BFS_BATCH_SIZE	50	Pages per batch in BFS.
CACHE_TTL	86400	Default cache TTL (24h).
WIKIPEDIA_MAX_WORKERS	10	Thread pool for Wikipedia API.
CELERY_TASK_SOFT_TIME_LIMIT	300	Soft timeout (5m).
CELERY_TASK_TIME_LIMIT	600	Hard timeout (10m).