Diagnosing and Resolving Silent PostgreSQL Connectivity Failures on Fly.io

Executive Summary

An application deployed on Fly.io that passes basic health checks yet fails during user authentication with a 401 Unauthorized error points to a silent but critical failure in its dependency chain. When the root cause is an inability to connect to the attached Fly Postgres database, the problem lies not in a catastrophic crash but in a subtle breakdown of communication or configuration. This report provides an exhaustive analysis of the potential causes for such a silent failure and a prioritized plan for diagnosis and resolution.

The most probable causes fall into four distinct categories. First, intermittent failures in Fly.io's internal private network DNS can render the database temporarily unreachable, a condition that standard application logic may not handle gracefully. Second, misconfigurations within the DATABASE_URL secret—ranging from simple typos to a fundamental misunderstanding of its availability during different deployment stages—are a frequent source of error. Third, the architecture of Fly Postgres, which utilizes the pgbouncer connection pooler, introduces a layer of abstraction that can conflict with common application-level database practices, such as the use of prepared statements.

Finally, the "silent" nature of this failure is a direct consequence of inadequate health checks. A standard health check verifies only the liveness of the application process itself, not its ability to communicate with critical dependencies. Without a robust health check that actively validates database connectivity, the Fly.io platform will continue to route traffic to a functionally impaired application instance, leading to user-facing errors that are disconnected from any obvious platform-level failure. Resolving this issue requires a systematic approach that validates each of these potential failure points, followed by the implementation of more resilient application architecture.

Part I: In-Depth Analysis of Potential Failure Points

Section 1: Private Networking and DNS Resolution Failures

The foundation of app-to-database communication on Fly.io is its private networking layer. While robust, its abstractions can introduce unique and intermittent failure modes that align with the symptoms of a silent connection loss.

1.1 The Fly.io 6PN Architecture: How App-to-DB Communication Works

Communication between a deployed application and its associated Fly Postgres instance occurs over a private IPv6 network, known as a 6PN (6-Private-Network). Every application within a Fly.io organization is connected to this mesh of WireGuard tunnels by default, ensuring that database traffic is not exposed to the public internet.

This internal communication relies on a specialized DNS resolver provided by the platform. This resolver is responsible for mapping hostnames ending in the .internal top-level domain to the correct private IPv6 addresses of the target machines. When a Fly Postgres cluster is created and attached to an application, the DATABASE_URL secret is populated with a connection string that uses this .internal hostname (e.g., my-db-app.internal). This architecture ensures that the connection is secure and internal by default. A failure in this system points to a breakdown within this private communication path, not an external network or firewall issue.

1.2 Common Pitfall: Intermittent DNS Resolution Glitches (could not translate host name)

While the .internal DNS system is generally reliable, it is not infallible. A recurring issue documented in community forums is the sudden onset of errors like could not translate host name "top2.nearest.of.my-app.internal" to address. This error explicitly indicates a failure of the internal DNS resolver to find the requested database host.
This behavior suggests a service discovery failure rather than a network partition. The underlying network connectivity between the app and database VMs remains intact, but the mechanism for discovering the database's IP address has failed. This can happen if the database machine's registration with the platform's DNS service becomes stale or corrupted. Evidence from platform support indicates that the prescribed solution is to manually stop and restart the database machine(s) using fly machine stop <machine_id> followed by fly machine start <machine_id>. This action forces the machine to re-register itself with the DNS service, typically resolving the issue.
Further evidence points to the possibility of inconsistent DNS resolution even within a single machine's environment. One user reported that their main application process could not resolve a .flycast address, while a manual connection attempt from an interactive Node.js REPL, initiated via an SSH session into the *same VM*, succeeded. This points to subtle environmental differences between the running application process and an interactive shell.
The implication for application design is significant. The dynamic nature of the platform means that service discovery can be transiently unavailable. Applications must be architected with this possibility in mind. Connection logic should not assume that a database address is perpetually resolvable; instead, it should incorporate robust retry mechanisms, ideally with exponential backoff, to handle temporary DNS lookup failures gracefully. Without such resilience, a momentary DNS glitch can translate into a persistent connection failure for the lifetime of the application process.

1.3 Common Pitfall: IPv6 vs. IPv4 Connectivity Issues

The Fly.io 6PN is an IPv6-native network. Consequently, all .internal hostnames resolve exclusively to IPv6 addresses. This can become a point of failure if any component in the application's stack—from the base Docker image to the database client library—lacks proper IPv6 support or is misconfigured.
If a database client library or the underlying operating system's resolver attempts to look up an A record (IPv4) for the .internal hostname, the lookup will fail. This can manifest as a standard DNS resolution error, such as ENOTFOUND in a Node.js application, even though the host is perfectly resolvable via an AAAA record (IPv6) lookup. Community discussions have highlighted this as a potential problem for various frameworks, where explicit configuration for IPv6 was required to resolve connectivity issues. The root cause in these scenarios is not a failure of the Fly.io network but a protocol mismatch originating from the client-side environment.

Section 2: Secrets Management and DATABASE_URL Integrity

Configuration errors are among the most common and difficult-to-diagnose sources of silent failures. A single incorrect character or a misunderstanding of platform mechanics related to the DATABASE_URL can prevent database connectivity without causing an obvious application crash.

2.1 Anatomy of the Fly.io DATABASE_URL

The DATABASE_URL is a standard connection string URI with a specific format: postgres://{username}:{password}@{hostname}:{port}/{database}. Each component must be correct for a connection to succeed.

- **username:password**: The credentials for the database user.
- **hostname**: The internal address of the Postgres cluster. This can take several forms, including a simple app-name.internal, a region-aware top2.nearest.of.app-name.internal, or a Flycast address like app-name.flycast.
- **port**: The port on which the database service is listening. For connections routed through pgbouncer, this is typically 5432.
- **database**: The specific database name within the Postgres cluster to connect to.

A mistake in any of these components will lead to a connection failure.

2.2 Common Pitfall: Malformed Connection Strings

Simple human error is a frequent culprit. A developer resolved a persistent connection issue after discovering a typo in the database name within the DATABASE_URL—a hyphen had been used instead of an underscore.

A more subtle failure mode occurs when the application successfully connects to the Postgres server but targets the *wrong database*. When a user connects to their cluster via fly pg connect, they are often placed in the default postgres database. If they run migrations from this session, the application's tables will be created there. However, the DATABASE_URL is typically configured to point to an application-specific database (e.g., my_app_db).

In this scenario, the application starts, the TCP connection to the Postgres server on port 5432 is successful, and the session is established with the my_app_db database. The application appears healthy. However, when it attempts to query a session table, the query fails with a "relation does not exist" error because the tables are in the postgres database instead. If the session storage library handles this SQL error gracefully (e.g., by returning null), the authentication logic will fail, correctly returning a 401 Unauthorized response to the end-user. The application never crashes, and the failure is entirely silent from a platform perspective.

2.3 Common Pitfall: Build-Time vs. Runtime Availability of Secrets

A critical architectural detail of the Fly.io platform is the distinction between the build environment and the runtime environment. Secrets set via fly secrets set, including the DATABASE_URL, are injected into the application's environment only at *runtime*. They are not available during the Docker image build process.

This design choice, which prevents secrets from being baked into container images, is a common trap for developers. If a command that requires database access, such as npx prisma migrate deploy, is included in the Dockerfile, it will fail because the DATABASE_URL environment variable is not defined at that stage.

This forces a more robust deployment pattern. Database migrations and other release tasks that depend on runtime secrets must be executed using the [deploy].release_command directive in the fly.toml configuration file. This command is executed in a temporary machine after the new image is built but before the new application version is deployed to serve traffic. This temporary machine has the new code and full access to runtime secrets, ensuring that the database schema is updated and compatible before the new code goes live. While a frequent source of initial deployment failure, this mechanism guides developers toward a safer and more reliable continuous deployment strategy.

2.4 Common Pitfall: SSL/TLS Parameter (sslmode) Misconfiguration

The traffic within Fly.io's private 6PN is already encrypted by WireGuard. As a result, Fly Postgres does not enable or require TLS for internal connections by default. However, many modern PostgreSQL client libraries default to requiring or preferring a TLS connection (sslmode=require).
This mismatch will cause the connection attempt to be rejected at the protocol level. The solution is often to explicitly disable client-side TLS by appending ?sslmode=disable to the DATABASE_URL connection string. This is a silent configuration failure, as it does not cause the application to crash but simply prevents it from establishing a database session. Conversely, when connecting to an external database that *does* enforce TLS, this parameter must be set correctly (e.g., sslmode=require), and sometimes requires additional options to handle certificate verification.

Section 3: Fly Postgres Internals: pgbouncer and Connection Pooling

Applications on Fly.io do not connect directly to a PostgreSQL process. Instead, they communicate through pgbouncer, a lightweight connection pooler. This intermediate layer is crucial for performance and scalability but also introduces its own set of potential issues that can be opaque to the application developer.

3.1 Understanding the Role of pgbouncer

Both managed and unmanaged Fly Postgres offerings use pgbouncer to manage a pool of connections to the underlying database server. This allows the platform to serve a large number of concurrent application clients with a much smaller number of actual PostgreSQL backend processes, significantly reducing memory and CPU overhead.
pgbouncer operates in different modes, which dictate how a server connection is allocated to a client. The most common modes are session (the default and safest, where a client holds a server connection for its entire session) and transaction (where a server connection is only allocated for the duration of a single transaction). This distinction is critical, as the choice of mode has profound implications for application compatibility.

3.2 Common Pitfall: Connection Pool Saturation and Timeouts

pgbouncer has finite resources, defined by configuration parameters like max_client_conn (the total number of incoming application connections it will accept) and default_pool_size (the number of actual connections to the backend Postgres server per database). If all available server connections are in use, new client requests are queued.
This can lead to silent failures if an application's request for a connection from the pool times out

while waiting in the queue. The application process itself does not crash; it simply fails to complete a database operation within its configured timeout period. This is particularly common during deployments, when a rolling update strategy can temporarily double the number of running application instances, all competing for the same limited pool of database connections. This can trigger a cascade of timeouts and errors like remaining connection slots reserved. From the user's perspective, this might manifest as a 502 Bad Gateway or, in the case of a failed session lookup, a 401 Unauthorized.

3.3 Common Pitfall: pgbouncer Mode Incompatibility (The Prepared Statements Trap)

The most insidious pgbouncer-related failures stem from incompatibilities between its pooling mode and application-level database features. The highly efficient transaction pooling mode achieves its performance by being stateless between transactions. This fundamentally breaks any PostgreSQL feature that relies on session state. Incompatible features include session-level advisory locks, temporary tables, and, most critically, session-level prepared statements.
Many database client libraries, including Node.js's node-postgres (pg) and Elixir's Ecto, use protocol-level prepared statements by default to optimize query performance and prevent SQL injection. When an application using these libraries connects to pgbouncer in transaction mode, a race condition can occur. The client might send a PREPARE command on one backend connection, but the subsequent EXECUTE command, being in a new transaction, might be routed by pgbouncer to a different backend connection where the statement was never prepared. This results in errors like FATAL 08P01 (protocol_violation) or ERROR: prepared statement "..." does not exist.
This is a prime candidate for a silent failure. The error occurs deep within the database driver's protocol handling and may not be surfaced in a way that crashes the main application process. The application continues to run, but every database query that relies on a prepared statement will fail, rendering parts of the application non-functional. The resolution is either to switch pgbouncer to the less-scalable but safer session mode or to configure the client library to disable the use of named prepared statements. Newer versions of pgbouncer (1.21+) have introduced support for prepared statements in transaction mode, but this requires careful configuration of the max_prepared_statements setting on the server.

Mode	How It Works	Compatible Features	Incompatible Features	Recommended Use Case
Session Pooling	A server connection is assigned to a client for its entire session.	All PostgreSQL features, including transactions, prepared statements, advisory locks, and temporary tables.	None.	Applications that rely heavily on session-state features or when compatibility is prioritized over maximum connection scaling.
Transaction Pooling	A server connection is assigned to a client only for the duration of a single transaction.	Basic DML/DDL, NOTIFY, protocol-level prepared statements (with modern pgbouncer	Session-level prepared statements (PREPARE/DEALLOCATE), session-level	High-throughput applications with many short-lived transactions that do not rely on session state.

Mode	How It Works	Compatible Features	Incompatible Features	Recommended Use Case
		versions).	advisory locks, SET/RESET commands, WITH HOLD cursors.	Provides a balance of performance and compatibility.
Statement Pooling	A server connection is assigned for a single statement and returned immediately.	Autocommit-style, single-statement operations.	Multi-statement transactions are disallowed. All features incompatible with transaction pooling are also incompatible here.	Highly specialized use cases, such as PL/Proxy, where "autocommit" behavior is enforced. Not suitable for general application use.

Section 4: The Illusion of Health: Why Basic Health Checks Deceive

The central mystery of a silent failure—an application that appears healthy to the platform but is non-functional for users—is almost always explained by inadequate health checks.

4.1 Deconstructing the Standard Fly.io Health Check

By default, Fly.io health checks are simple probes designed to determine if a machine should receive traffic from the Fly Proxy. These are typically configured in the fly.toml file and come in two primary forms :

- **TCP Checks ([[services.tcp_checks]])**: These verify that a process is listening on a specific TCP port. A successful check means only that the port is open.
- **HTTP Checks ([[services.http_checks]])**: These send an HTTP request to a specified path (e.g., /) and expect a 2xx status code in response.

A standard health check, therefore, only confirms that the web server process (e.g., Express.js) has successfully started and bound to its port. It provides zero visibility into the status of critical downstream dependencies, such as the PostgreSQL database. The application can be completely unable to communicate with its database, yet it will continue to pass these basic checks and be considered "healthy" by the platform.

4.2 Architecting a Robust, Dependency-Aware Health Check Endpoint

To prevent silent failures, a health check must be a comprehensive, end-to-end test of the application's critical path. For a database-driven application, this means the health check endpoint must actively validate database connectivity.
A robust health check endpoint, often exposed at a path like /healthz, should perform the following sequence of operations:

1. Attempt to acquire a connection from the application's database connection pool.
2. Execute a simple, non-intrusive, and fast query, such as SELECT 1.
3. Release the connection back to the pool.
4. If all steps succeed, return an HTTP 200 OK status.
5. If any step fails (e.g., the connection acquisition times out, or the query returns an error), it

must return a non-2xx status code, such as 503 Service Unavailable.

When this logic is implemented, a database connectivity issue will cause the health check to fail. The Fly Proxy will detect the failure and immediately stop routing traffic to the unhealthy machine. This makes the failure loud and immediate, either by routing users to a remaining healthy instance or by serving a clear error page, thus preventing the confusing 401 symptom and making the problem's scope immediately apparent to operators.

4.3 Utilizing machine_checks for Pre-Deployment Validation

Beyond runtime health checks, Fly.io provides a mechanism for validating a new release *before* it is deployed: machine_checks. Configured in fly.toml, a machine_check runs a custom command inside an ephemeral machine created with the new application image. This check is explicitly designed for validating complex readiness scenarios, including "confirming database connectivity".

If the machine_check command exits with a non-zero status code, the entire deployment is halted, preventing a broken release from reaching production. This is a powerful, proactive tool. For instance, a machine_check could be a script that uses the runtime DATABASE_URL to connect to the database and verify that the schema version is compatible with the new code. This catches configuration errors and database-related issues at the earliest possible stage in the deployment pipeline.

Section 5: Latent Platform and Resource Constraints

Sometimes, the source of a silent failure is not the application or the network path but a resource constraint on the Postgres VM itself. These issues can cause the database to enter a degraded state that is not immediately obvious.

5.1 Common Pitfall: Full Storage Volume Leading to Read-Only State

As a protective measure to prevent data loss and corruption, a Fly Postgres instance will automatically switch to a read-only mode if its attached storage volume approaches its capacity. When this happens, the Postgres server continues to run and accept connections.

This is a classic silent failure scenario. The application can still connect to the database, and any SELECT queries (such as reading configuration or product data) will succeed. The basic health checks will pass. However, any attempt to perform a write operation—such as inserting a new record into a session table—will be rejected by the database with a read-only transaction error. If the application's error handling for this specific case is not robust, it will simply fail the operation, leading to an authentication failure and a 401 response, all while the application process itself remains stable. The issue is only resolved by extending the volume size via flyctl commands.

5.2 Common Pitfall: Out-of-Memory (OOM) Events

Postgres can be memory-intensive, and crashes on the platform are frequently attributed to out-of-memory (OOM) errors. When the Postgres VM runs out of memory, the kernel's OOM killer will terminate the Postgres process to preserve the stability of the machine. The Fly.io platform will then automatically restart the process.

During the window between the crash and the successful restart—which can be several minutes—the database is completely unavailable. All connection attempts from the application will fail. If the application's connection logic has a short timeout and no built-in retry mechanism, these attempts will fail silently. The application process continues to run, but for the duration of the database outage, it is functionally useless for any operation requiring a database connection. Any user attempting to authenticate during this window will receive a 401 error. This type of intermittent outage can be difficult to diagnose without inspecting the Postgres app's logs and metrics for evidence of crash loops or OOM events.

Part II: A Prioritized Diagnostic and Resolution Plan

Section 6: Recommended Diagnostic Workflow

This section provides a systematic, step-by-step workflow to isolate and identify the specific cause of the database connectivity failure. The steps are ordered from least to most invasive, starting with simple verification and progressing to direct inspection.

Step 1: Verify Secrets and Environment at Runtime

The first step is to confirm that the application has the correct DATABASE_URL. Local configuration files or build-time issues can create a discrepancy between what is expected and what is actually present in the runtime environment.

- **Action:** SSH into a running application machine and print the DATABASE_URL environment variable as the process sees it.
  ```
  fly ssh console -a your-app-name -C "env | grep DATABASE_URL"
  ```
- **Rationale:** This command bypasses all local configurations and directly reveals the secret that has been injected into the runtime environment. It is the single source of truth for the connection string.
- **Verification:** Carefully examine the output for any errors:
  - Is the hostname (<db-app-name>.internal) correct?
  - Is the database name at the end of the string correct? Check for common typos like hyphens vs. underscores.
  - Are there any unexpected characters or formatting issues?

Step 2: Test Network Path and DNS from Inside the App VM

If the DATABASE_URL is correct, the next step is to verify the network path and DNS resolution from the application's perspective.

- **Action:** Open an interactive shell inside the application VM.
  ```
  fly ssh console -a your-app-name
  ```
- **Action:** From within the shell, use dig to test internal DNS resolution for the database host.
  ```
  # Inside the SSH session
  dig aaaa your-db-app-name.internal
  ```
- **Rationale:** This command directly queries Fly.io's internal DNS server. If it returns one or more AAAA records with IPv6 addresses, DNS is working. If it fails or times out, it confirms a DNS resolution problem as described in Section 1.2.

- **Resolution (if dig fails):** The immediate fix is to restart the Postgres machine(s) to force re-registration with the DNS service.
  ```
  # In your local terminal
  fly machine list -a your-db-app-name # Get machine IDs
  fly machine stop <machine_id> -a your-db-app-name
  fly machine start <machine_id> -a your-db-app-name
  ```
- **Action (if dig succeeds):** Attempt a manual database connection from within the VM. For a Node.js application, this can be done using the Node.js REPL. First, get the full DATABASE_URL from Step 1.
  ```
  # Inside the SSH session
  node
  ```
  Then, inside the Node.js REPL:
  ```
  const { Client } = require('pg');
  const client = new Client({ connectionString:
  "paste_your_full_database_url_here" });
  client.connect().then(() => console.log('Connection
  successful!')).catch(err => console.error('Connection error:',
  err)).finally(() => client.end());
  ```
- **Rationale:** This test isolates the problem. If this manual connection succeeds, the network path, DNS, credentials, and Postgres server are all functioning correctly. The problem must lie within the main application's code, its specific process environment, or its connection pooling logic. If this manual connection fails, the problem is confirmed to be with the DATABASE_URL itself, the Postgres server, or a network-level issue.

Step 3: Inspect Postgres App Health and Resources

If direct connection attempts fail, the focus should shift to the health of the Postgres cluster itself.

- **Action:** Check the status of all machines in the Postgres cluster.
  ```
  fly status --all -a your-db-app-name
  ```
- **Verification:** Look for any machines in a failed state, a high number of RESTARTS, or a STATUS of running (readonly). A readonly status is a strong indicator of a full disk volume.
- **Action:** If a machine is in a readonly state, SSH into it and check the disk usage.
  ```
  fly ssh console -a your-db-app-name
  # Inside the SSH session
  df -h
  ```
- **Action:** Check the logs for the Postgres application, looking for crash reports, OOM (out-of-memory) errors, or connection error messages.
  ```
  fly logs -a your-db-app-name
  ```
- **Rationale:** These commands provide direct insight into the operational health and resource constraints of the database server, addressing the potential causes outlined in Section 5.

Step 4: Enhance Application-Level Connection Logging

The default error messages from database drivers are often generic (e.g., "Connection terminated unexpectedly") and lack the detail needed for effective debugging. The final diagnostic step is to enhance the application's own logging to capture more granular error information.

- **Action (for node-postgres):** Attach an error listener to the connection pool. This is crucial for capturing errors on idle clients, which can occur due to network partitions or database restarts and would otherwise crash the Node.js process.
```
const { Pool } = require('pg');
const pool = new Pool({ connectionString: process.env.DATABASE_URL
});

pool.on('error', (err, client) => {
  console.error('Unexpected error on idle client', err);
  // It's often recommended to let the pool handle closing the
client
});
```
- **Action:** Wrap individual connection and query calls in try...catch blocks and log the entire error object, not just the message. The full object often contains valuable context like error codes.
```
async function getSession(sessionId) {
  let client;
  try {
    client = await pool.connect();
    const result = await client.query('SELECT * FROM sessions
WHERE id = $1', [sessionId]);
    return result.rows;
  } catch (err) {
    console.error('Error executing getSession query:', err); //
Log the full error
    throw err;
  } finally {
    if (client) {
      client.release();
    }
  }
}
```
- **Action (Advanced):** Integrate a dedicated monitoring library like pg-monitor to automatically log all database events, including connections, disconnections, queries, and errors, providing a complete trace of the application's interaction with the database.
- **Rationale:** By capturing detailed, application-aware error information, it becomes possible to distinguish between a DNS failure (ENOTFOUND), a TLS handshake error, an authentication failure, a pgbouncer protocol violation, or a simple query timeout.

Command	Purpose	Example Usage
fly logs -a <app-name>	View real-time and historical logs for an application.	fly logs -a my-shopify-app
fly ssh console -a <app-name>	Open an interactive SSH shell into a running application VM.	fly ssh console -a my-shopify-app
fly ssh console -C "<command>"	Execute a single command inside a VM without opening an interactive shell.	fly ssh console -a my-app -C "env"
fly dig aaaa <host> -a <app-name>	Perform a DNS lookup for an IPv6 address against Fly.io's internal DNS.	fly dig aaaa my-db.internal -a my-app
fly status --all -a <app-name>	Display the status, region, and health of all machines for an app.	fly status --all -a my-db-app
fly machine status <id>	Get detailed information and recent events for a specific machine.	fly machine status 123456789
fly pg connect -a <pg-app-name>	Open a psql shell connected to the primary Postgres instance.	fly pg connect -a my-db-app
fly proxy <local>:<remote> -a <app>	Forward a local port to a port on a remote application VM.	fly proxy 54321:5432 -a my-db-app

Section 7: Advanced Troubleshooting and Long-Term Solutions

Once the immediate issue is resolved, implementing more resilient architectural patterns is essential to prevent future silent failures.

7.1 Implementing a Comprehensive Health Check: A Code-Level Example

The most effective long-term solution is to create a dependency-aware health check endpoint. This ensures that the platform's view of application health accurately reflects its ability to perform its core function.
Here is an example implementation for a Node.js/Express application using a shared pg connection pool:
Application Code (server.js or similar):
```
const express = require('express');
const { Pool } = require('pg');

const app = express();
const pool = new Pool({ connectionString: process.env.DATABASE_URL });

//... your other application routes...

app.get('/healthz', async (req, res) => {
  let client = null;
  try {
    // Attempt to get a client from the pool.
    // Set a short timeout to prevent the health check from hanging.
    client = await pool.connect();

    // Execute a simple, fast query.
    await client.query('SELECT 1');

    // If successful, the database is reachable.
    res.status(200).send('OK');
  } catch (error) {
    // If any part of the check fails, log the error and return 503.
    console.error('Health check failed:', error);
    res.status(503).send('Service Unavailable');
  } finally {
    // Ensure the client is always released back to the pool.
    if (client) {
      client.release();
    }
  }
});

const port = process.env.PORT |

| 8080;
app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

Configuration (fly.toml): This configuration directs the Fly Proxy to use the new /healthz endpoint for its checks. The grace_period is critical, as it gives the application time to start up and establish its initial database connections before the checks begin.

```
# fly.toml

app = "your-app-name"
primary_region = "iad"

#... other configurations...

[[services]]
  internal_port = 8080
  protocol = "tcp"

  [[services.http_checks]]
    interval = "15s"
    timeout = "2s"
    method = "GET"
    path = "/healthz"
    grace_period = "10s" # Adjust based on your app's startup time
```

7.2 Proactive Monitoring and Alerting

Fly.io exposes a rich set of Prometheus metrics for all applications, including specialized metrics for Fly Postgres and volumes. Instead of reacting to outages, these metrics allow for proactive monitoring of database health.
Key metrics to monitor include:

- **fly_volume_used_pct**: The percentage of disk space used on the Postgres volume. Alerting when this exceeds a threshold (e.g., 80%) can prevent the database from entering a read-only state.
- **fly_instance_memory_**: Memory usage on the Postgres VM. Spikes or a consistently high usage pattern can indicate an impending OOM event.
- **Postgres-specific metrics (pg_*)**: Metrics exported by postgres_exporter can provide insight into the number of active connections, transaction rates, and query performance.

By integrating these metrics with a monitoring solution like Grafana and configuring alerts, operational teams can identify and address resource constraints long before they escalate into a service-disrupting silent failure.

Works cited

1. Private Networking · Fly Docs - Fly.io, https://fly.io/docs/networking/private-networking/ 2. PostgreSQL · Fly Docs - Fly.io, https://fly.io/docs/deep-dive/postgresql/ 3. Healthcare apps on Fly · Fly Docs - Fly.io, https://fly.io/docs/about/healthcare/ 4. Connect From a Fly App · Fly Docs - Fly.io, https://fly.io/docs/postgres/connecting/connecting-internal/ 5. Connection Error on Node.js App with Fly.io Postgres database, https://community.fly.io/t/connection-error-on-node-js-app-with-fly-io-postgres-database/10150 6. Cannot connect Node.js app to Postgres db - Build debugging - Fly.io Community, https://community.fly.io/t/cannot-connect-node-js-app-to-postgres-db/6631 7. No access to DATABASE_URL hostname - Fly.io, https://community.fly.io/t/no-access-to-database-url-hostname/18944 8. Unable to connect to Fly Postgres via node JS application - Questions / Help, https://community.fly.io/t/unable-to-connect-to-fly-postgres-via-node-js-application/22122 9. Facing issue on shopify remix app hoisting with postgres db in fly.io, https://community.shopify.com/t/facing-issue-on-shopify-remix-app-hoisting-with-postgres-db-in-fly-io/385245 10. DB Connection issues - postgres - Fly.io Community, https://community.fly.io/t/db-connection-issues/5801 11. How to setup and use PGBouncer with Fly Postgres - #35 by sudhir.j - Fly.io Community, https://community.fly.io/t/how-to-setup-and-use-pgbouncer-with-fly-postgres/3035/35 12. Connection refused between Fly Express app and Fly Postgres ..., https://community.fly.io/t/connection-refused-between-fly-express-app-and-fly-postgres/18667 13. External Connections to Fly Postgres (Update: Where's my data???) - Fly.io Community, https://community.fly.io/t/external-connections-to-fly-postgres-update-wheres-my-data/12736 14. Can't find DATABASE_URL during deployment - Questions / Help ..., https://community.fly.io/t/cant-find-database-url-during-deployment/7719 15. Seeking advice about securing postgres on Fly - Questions / Help, https://community.fly.io/t/seeking-advice-about-securing-postgres-on-fly/7861 16. How to connect from Fly App to Fly Postgres using `sslmode=require` - Fly.io Community, https://community.fly.io/t/how-to-connect-from-fly-app-to-fly-postgres-using-sslmode-require/24034 17. Problem connecting to External Database - Phoenix - Fly.io Community,

https://community.fly.io/t/problem-connecting-to-external-database/22779 18. Cluster configuration options · Fly Docs - Fly.io, https://fly.io/docs/mpg/configuration/ 19. Node.js + PostgreSQL: The Simple Trick to Effortlessly Scale 10,000+ Connections | by Rajat Gupta | Medium,
https://medium.com/@rajat29gupta/node-js-postgresql-the-simple-trick-to-effortlessly-scale-10-000-connections-312c3079d362 20. Managed Postgres · Fly Docs - Fly.io, https://fly.io/docs/mpg/ 21. Create and Connect to a Managed Postgres Cluster - Fly.io,
https://fly.io/docs/mpg/create-and-connect/ 22. Using PgBouncer - NetApp Instaclustr,
https://www.instaclustr.com/support/documentation/postgresql-add-ons/using-pgbouncer/ 23. Features - PgBouncer, https://www.pgbouncer.org/features.html 24. PgBouncer config,
https://www.pgbouncer.org/config.html 25. Fly rollouts and PgBouncer limits - postgres - Fly.io Community, https://community.fly.io/t/fly-rollouts-and-pgbouncer-limits/24028 26. How to setup and use PGBouncer with Fly Postgres - #27 by sudhir.j - Fly.io Community,
https://community.fly.io/t/how-to-setup-and-use-pgbouncer-with-fly-postgres/3035/27 27. 502 Error: Node.js app with Postgres on Fly.io not working as expected - Build debugging,
https://community.fly.io/t/502-error-node-js-app-with-postgres-on-fly-io-not-working-as-expected/14602 28. PgBouncer is useful, important, and fraught with peril - JP Camara,
https://jpcamara.com/2023/04/12/pgbouncer-is-useful.html 29. Using pgBouncer on DigitalOcean with Node.js pg Pool and Kysely – Can They Coexist?,
https://www.reddit.com/r/PostgreSQL/comments/1jvb4z9/using_pgbouncer_on_digitalocean_with_nodejs_pg/ 30. PgBouncer: The one with prepared statements - DEV Community,
https://dev.to/neon-postgres/pgbouncer-the-one-with-prepared-statements-198i 31. Persistent Postgrex Protocol/Connection Errors on Managed Postgres - Fly.io Community,
https://community.fly.io/t/persistent-postgrex-protocol-connection-errors-on-managed-postgres/25676 32. Prepared Statements in Transaction Mode for PgBouncer | Crunchy Data Blog,
https://www.crunchydata.com/blog/prepared-statements-in-transaction-mode-for-pgbouncer 33. Boosting Postgres Performance With Prepared Statements and PgBouncer's Transaction Mode | TigerData,
https://www.tigerdata.com/blog/boosting-postgres-performance-with-prepared-statements-and-pgbouncers-transaction-mode 34. Health Checks · Fly Docs,
https://fly.io/docs/reference/health-checks/ 35. Seamless Deployments on Fly.io,
https://fly.io/docs/blueprints/seamless-deployments/ 36. Unable to connect to postgres via fly postgres connect, or proxy. - Questions / Help - Fly.io,
https://community.fly.io/t/unable-to-connect-to-postgres-via-fly-postgres-connect-or-proxy/9257 37. Troubleshoot your deployment · Fly Docs - Fly.io,
https://fly.io/docs/getting-started/troubleshooting/ 38. Postgres connection terminating unexpectedly - Build debugging - Fly.io Community,
https://community.fly.io/t/postgres-connection-terminating-unexpectedly/8720 39. pg-monitor - NPM, https://www.npmjs.com/package/pg-monitor 40. Metrics on Fly.io · Fly Docs,
https://fly.io/docs/monitoring/metrics/