MyCount v2 Delivery Report

Overview

MyCount remains a full-stack expense-sharing tool built on FastAPI, PostgreSQL, React, and Docker. This report summarizes the changes introduced after release/v1, focusing on production hardening, delivery automation, and observability. The goal was to make the service deployable, monitorable, and testable with minimal manual steps.

Highlights Compared to Version 1

- Production hardening: Added resilient database startup, explicit health checks, and better error handling. The database session now retries connections before failing, and a dedicated health endpoint verifies the database rather than just the process state.
- Observability: Instrumented HTTP traffic with Prometheus counters and histograms, exposed a native metrics endpoint, and provisioned Prometheus plus Grafana containers with a prebuilt backend dashboard.
- Delivery automation: Introduced a GitHub Actions workflow that gates merges on backend coverage, builds all service images, and redeploys Azure Container Apps for frontend, backend, Prometheus, and Grafana.
- Configuration safety: Normalized database URLs for Azure, enforced SSL by default, and refreshed environment examples so local and cloud deployments stay aligned.
- Testing discipline: Added a reusable Makefile target that produces coverage.xml, enforces a coverage floor, and feeds the CI job. Coverage artifacts are now checked into the repo so the pipeline can validate them without extra tooling.

What Changed in Detail

Backend resilience and configuration

- Database connection retries now happen at process start (backend/app/db/session.py). The engine attempts three connections with waits before failing, logging each attempt so container restarts are visible.
- Settings (backend/app/core/config.py) normalize database URLs, translate postgres schemes to psycopg2, and append sslmode=require when missing. This makes the same configuration usable locally and against Azure Database for PostgreSQL.
- Health and metrics routes live in backend/app/api/system.py. Health runs a lightweight SELECT 1 through the pooled session, while metrics returns Prometheus exposition format via backend/app/core/metrics.py.
- Integrity errors now return a clear 400 response instead of bubbling SQL errors to clients (backend/app/main.py).

Data and bootstrapping

- backend/app/db/init.sql now drops existing objects before recreating them so repeated docker-compose runs stay idempotent. The script keeps expense, expense_split, group, group_invite, group_members, and user tables aligned with the ORM schema.
- The group_invite table gains created_at, expires_at, and used flags to support one-time invite tokens consistently across environments.

Runtime and images

- backend/Dockerfile installs build-essential and libpq headers so psycopg2 builds reliably in slim images, and the container exposes a fixed port 8000 for predictable Azure routing.
- docker-compose.yml now includes Prometheus and Grafana services wired to the backend metrics endpoint, plus persisted volumes for database, Prometheus data, and Grafana state.
- frontend and backend .env.example files were refreshed to document required variables for local and cloud usage.

Developer workflow and tests

- backend/Makefile adds test, coverage, clean, and dev targets. The coverage target enforces a minimum threshold, emits HTML and XML reports, and echoes their locations for quick inspection.
- coverage.xml is generated in CI and stored so the workflow can assert that total coverage stays above 80 percent before images are built or pushed.

CI/CD Pipeline

The GitHub Actions workflow at .github/workflows/deploy.yml now owns testing, image builds, and Azure rollouts. It triggers on pushes to master and develop. The workflow has two jobs: test-backend and build-and-push. The second job waits on the first to keep broken code from shipping.

Backend tests must pass with coverage above 80 percent for the second job to run. The job runs make coverage, then uses the coverage report to enforce the threshold. Excerpt:

```yaml
test-backend:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v4
      with:
        python-version: "3.11"
    - name: Install Backend Dependencies
      working-directory: backend
      run: pip install -r requirements.txt
    - name: Run Tests with Coverage
      working-directory: backend
      run: make coverage
    - name: Check Coverage > 80%
      working-directory: backend
      run: |
        coverage=$(coverage report | grep TOTAL | awk '{print $4}' | sed 's/%//')
        if (( $(echo "$coverage < 80" | bc -l) )); then
          exit 1
        fi
```

Delivery steps after the tests:

- Build and push four images to Docker Hub: mycount-frontend, mycount-backend, mycount-grafana, and mycount-prometheus. Each uses the local Dockerfiles and tags them with latest.
- Authenticate to Azure using a federated identity (secrets.AZURE_CREDENTIALS) and call az containerapp update for each running app. The backend updates mycount-backendv2, the frontend updates mycount-frontend, and monitoring services update mycount-prometheus. Grafana is built and pushed for parity even though it deploys separately.
- Secrets are injected only at runtime (Docker Hub credentials, Azure credentials), keeping repository contents clean.

Operationally, this pipeline moves the project from manual docker pushes in v1 to a gated, repeatable delivery path that always runs tests first, produces uniform images, and refreshes cloud workloads in one run.

Monitoring and Observability

Application instrumentation

Requests are now wrapped in MetricsMiddleware (backend/app/core/metrics.py). It records totals, latencies, and error counts labeled by method and route template, and it skips paths like /metrics and /health to reduce noise. Example implementation:

```python
REQUEST_COUNTER = Counter(
    "http_requests_total",
    "Total HTTP requests",
    ("method", "path", "status"),
)
REQUEST_LATENCY = Histogram(
    "http_request_duration_seconds",
    "HTTP request latency in seconds",
    ("method", "path", "status"),
)
REQUEST_ERRORS = Counter(
    "http_request_errors_total",
    "Total HTTP requests that resulted in an error",
    ("method", "path"),
)

class MetricsMiddleware(BaseHTTPMiddleware):
    def __init__(self, app, skip_paths=None):
        super().__init__(app)
        self.skip_paths = set(skip_paths or [])

    async def dispatch(self, request, call_next):
        path = resolve_path_template(request)
        if path in self.skip_paths:
            return await call_next(request)

        method = request.method
        start_time = time.perf_counter()
        status_code = "500"
        error_recorded = False

        try:
            response = await call_next(request)
            status_code = str(response.status_code)
            return response
        except Exception:
            REQUEST_ERRORS.labels(method, path).inc()
            error_recorded = True
            raise
        finally:
            elapsed = time.perf_counter() - start_time
            REQUEST_COUNTER.labels(method, path, status_code).inc()
            REQUEST_LATENCY.labels(method, path, status_code).observe(elapsed)
            if not error_recorded and int(status_code) >= 500:
                REQUEST_ERRORS.labels(method, path).inc()
```

The middleware is registered in backend/app/main.py with skip_paths={"/metrics", "/health"} so scrape and health calls do not skew stats. Metrics are exposed at /metrics in Prometheus format.

Health checks

The new system router provides a database-aware health probe. It runs a lightweight query and returns 503 with an error log when the database is unavailable, preventing false positives from the container being up but the database being down. Implementation:

```python
@router.get("/health")
def health(db: Session = Depends(get_db), logger: Logger = Depends(get_request_logger)):
    try:
        db.execute(text("SELECT 1"))
        return {"status": "ok"}
    except SQLAlchemyError:
        logger.error("database health check failed")
        raise HTTPException(
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
            detail="Database unavailable",
        )
```

Prometheus

Prometheus is now part of docker-compose and the CI-built images. Scrape configuration targets the backend container on port 8000 at /metrics with 15-second intervals. The configuration is short and explicit:

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: backend
    metrics_path: /metrics
    static_configs:
      - targets:
          - backend:8000
```

Grafana

Grafana is provisioned automatically with a Prometheus data source and a curated dashboard stored at monitoring/grafana/dashboards/json/backend-overview.json. The dashboard tracks request rate, error rate, p95 latency, and request rate by path, which align with the metrics exported by the middleware. Datasource provisioning is defined in monitoring/grafana/datasources/datasource.yml to keep the setup reproducible. Grafana and Prometheus both gain persistent volumes so dashboards and time series survive container restarts.

Logging

Database connection attempts, health check failures, and integrity errors now produce structured logs that surface during container startup and runtime. This complements metrics by making failure causes visible without attaching a debugger.

Operations, Testing, and Usage

- Local run: docker-compose now starts the database, backend, frontend, Prometheus, and Grafana together. Volumes keep state between restarts, and backend init.sql resets schemas safely on rebuild.
- Tests: make test runs the suite, and make coverage enforces 80 percent minimums while emitting HTML and XML artifacts. The CI job reuses the same commands, eliminating drift between local runs and pipeline behavior.
- Configuration: settings.database_url automatically handles common Azure connection strings by converting postgres schemes to psycopg2 and forcing SSL. Default origins include both localhost and Vite defaults to simplify frontend integration.
- Runtime behavior: the backend image now contains necessary build tooling for psycopg2, exposes port 8000 explicitly, and uses uvicorn with fixed host and port for predictable Container App routing.

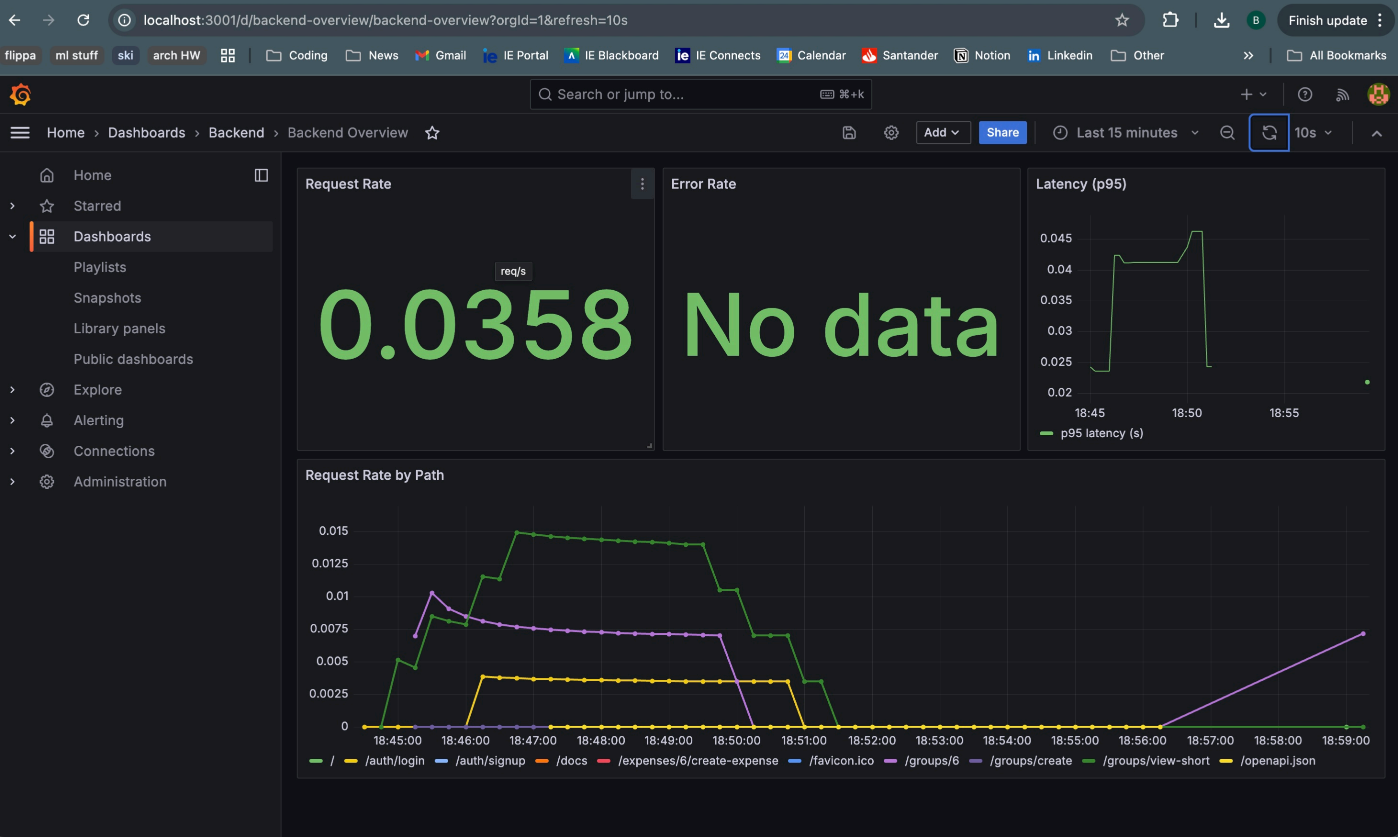
Conclusion

Compared to version 1, this release focuses on making the service trustworthy to deploy and observe: resilient startup, explicit health and metrics endpoints, test gating in CI, automated image builds, and pre-provisioned monitoring. Unfortunatley, I was unable to get the full application running on Azure. All of my services are working except for my backend (the most crucial one), here are some of the largest problems I encountered:

1. The first issue was the frontend failing to resolve my API_URL env variable. I tried to inject it at build-time instead (both when creating the image and the container), but still it was using the default host (localhost). In the end I just hardcoded the url.

2. The backend was unable to connect to the database. For this, I needed to manually re-construct the database URL in **config.py** to include psycopg2 and sslmode in the URL. Once this was fixed, I again was having trouble with the environment variables. The database host, password, etc. kept resolving to the default value, causing the application to raise a RunTime error at startup. In the end I hardcoded these values into the default-value as well.
3. The error I'm currently stuck on and haven't been able to fix is regarding the port that uvicorn is listening on. I've configured Azure to expect 8000 (TargetPort) and I've hardcoded the --port 8000 in my Dockerfile, but the app still tries to listen on port 32134.

In the future I'd like to fix all these the proper way, implementing strict networking rules and understanding why I got the errors that I did. For now, I've altered all sensitive hard-coded values and changed my database pw. Although not reachable online, you can still follow the local launch instructions using Docker to see the new endpoints as well as the Grafana setup.



Request Rate

req/s

0.0358

Error Rate

No data

Latency (p95)

Request Rate by Path