



SAMPLE REPORT. The system, company and findings below are fictional. This document shows the structure and depth of a CodeFlow code & architecture review.

Code & architecture review

Orders platform – Example Company (fictional)

Scope	Orders API (Python / Django REST), order worker (RabbitMQ consumer), PostgreSQL schema, CI pipeline
Size	3 repositories · ~18,000 lines of application code · 64 direct dependencies
Input	Git repositories (read-only), architecture notes, non-functional requirements, 45-minute team interview
Review period	5 working days
Confidentiality	Performed under NDA

1. Summary

The platform is in reasonable shape for its age. The code is readable, the domain model is clear and deployments are automated. Two issues need immediate attention: an access-control gap on the order endpoints and database credentials in the git history. Both can be fixed within days.

The main architectural risk is that the API and the worker share database tables without a defined contract. This will slow every future change to the order model.

Risk level	Findings	Meaning
HIGH	2	Fix now. Exposes data or creates a likely production incident.
MEDIUM	3	Plan within the next 1–2 sprints. Reliability, performance or growing cost of change.
LOW	2	Address during normal work. Maintainability and hygiene.

Recommended first three actions

1. Add an ownership check to all order endpoints, and a test for it (H-1).
2. Rotate the database credentials and move secrets to the deployment platform's secret store (H-2).
3. Add backoff and a dead-letter queue to the order worker (M-1).

2. Scope and method

Automated analysis (static analysis, secret scanning, dependency scanning), followed by manual review of the critical paths: order creation, payment confirmation, refunds and the worker pipeline. Findings were discussed with the team before this report was finalised.

Security checks included

- Common vulnerability classes (OWASP Top 10)
- Authentication, authorisation and access control
- Secrets in code, configuration and git history
- Known-vulnerable dependencies
- Error handling and logging of sensitive data

Not included

- Penetration testing
- Formal security audit or certification
- Compliance assessment (ISO 27001, SOC 2, PCI DSS, NIS2)
- Infrastructure and cloud configuration review

3. Findings overview

ID	Level	Finding	Area	Effort
H-1	HIGH	Orders can be read by any authenticated user	Security · access control	S
H-2	HIGH	Database credentials in git history	Security · secrets	S
M-1	MEDIUM	Worker retries without backoff or dead-letter queue	Reliability	M
M-2	MEDIUM	API and worker share tables without a contract	Architecture	L
M-3	MEDIUM	N+1 queries on the order list endpoint	Performance	S
L-1	LOW	Validation rules duplicated in three modules	Maintainability	M
L-2	LOW	Refund calculation has no automated tests	Testing	S

Effort: S = up to 1 day · M = 2–5 days · L = more than a week

4. Findings in detail

HIGH H-1 · Orders can be read by any authenticated user

Location	orders/views.py · OrderDetailView, OrderInvoiceView
Description	The views load an order by ID and check only that the user is logged in. Any customer can read another customer's order and invoice by changing the ID in the URL. This is broken access control, the most common category in the OWASP Top 10.
Recommendation	Filter the queryset by the requesting user's account in both views, and return 404 for orders that belong to someone else. Add a test that requests another account's order and expects 404.

```
def get_queryset(self):  
    return Order.objects.filter(account=self.request.user.account)
```

HIGH H-2 · Database credentials in git history

Location	config/settings_prod.py (removed in a later commit, still in history)
Description	The production database password was committed and later deleted. Deleting the file does not remove it from history, so anyone with repository access can read it.
Recommendation	Rotate the password first. Then load secrets from environment variables set in the deployment platform, and add a secret scanner (e.g. gitleaks) to CI so this cannot recur. Rewriting history is optional once the secret is rotated.

MEDIUM M-1 · Worker retries without backoff or dead-letter queue

Location	worker/consumer.py · handle_order_paid()
Description	Failed messages are requeued immediately and indefinitely. If the payment provider is down, the worker retries in a tight loop and blocks other messages behind it.
Recommendation	Use exponential backoff with a retry limit, then route the message to a dead-letter queue with an alert. Make the handler idempotent so a retried message cannot confirm an order twice.

MEDIUM M-2 · API and worker share tables without a contract

Location	orders_order, orders_payment tables · both repositories
Description	The worker writes directly to tables owned by the API, using its own copy of the models. Any schema change must be coordinated across two repositories, and the copies have already drifted (one field differs in length).
Recommendation	Make the API the only writer. The worker should publish an event or call an internal endpoint. As a first step, move the shared models into one package used by both.

MEDIUM M-3 · N+1 queries on the order list endpoint

Location	orders/serializers.py · OrderListSerializer
Description	Each order in the list triggers separate queries for its items and customer. A page of 50 orders runs over 100 queries.
Recommendation	Use select_related("customer") and prefetch_related("items") in the view's queryset, and add a query-count assertion to the endpoint test.

LOW L-1 · Validation rules duplicated in three modules

Location	orders/validators.py, billing/forms.py, api/v1/checks.py
Description	VAT number and postcode rules are implemented three times, with small differences.
Recommendation	Extract one shared validator module and cover it with table-driven tests.

LOW L-2 · Refund calculation has no automated tests

Location	billing/refunds.py · calculate_refund()
Description	Partial refunds with discounts follow several branches, none of which are tested.
Recommendation	Add unit tests for full, partial and discounted refunds before the next change to this code.

5. What is working well

- Clear domain model and consistent naming.
- Automated deployments with a working rollback.
- Good test coverage on order creation.

Appendix · Tools used

Static analysis (Semgrep, Ruff), secret scanning (gitleaks), dependency scanning (pip-audit, Trivy), query profiling (Django Debug Toolbar), plus manual review.

CodeFlow · [codeflow.gr](#) · Sample report – fictional system. This report reflects the code at the reviewed commit and is not a guarantee that the system is free of defects or vulnerabilities.