**Test Document**

**1 Document Introduction**

**1.1 Purpose of the Document**

This document aims to clearly define the scope, objectives, and boundaries of testing for the **Research and Application of Urban**

**Land Use Change Detection** project. It serves as a reference for all stakeholders—including developers, testers, project managers, and clients—to understand what will be tested, how it will be tested, and what is excluded from the testing process. By specifying the testing scope and approach, this document ensures that the testing activities are aligned with the project requirements and quality expectations, and helps to avoid misunderstandings or gaps in test coverage.

**1.2 Project Background**

This project aims to provide a platform for the analysis of the solar and thermal data in Shanghai. It mainly includes a heat map showing the grid data of Shanghai's solar and thermal aspects, and can be used to identify the key influencing factors of the solar and thermal dependent variables in the Excel data uploaded by users. It also provides solar and thermal predictions using the Kan model, as well as a comparison between the Kan model and other baseline models, aiming to help urban planning personnel analyze the livability level of a certain area in Shanghai through solar and thermal data.

**1.3 Scope of the Test Document**

**1.3.1 Service Modules**

* **api\_gateway:**
* API aggregation and routing
* Unified request/response format
* Service health check
* API endpoints for gateway
* **auth\_service:**
* User registration, login, logout
* JWT token generation and validation
* User CRUD and permission management
* Email verification and password reset
* API endpoints for authentication and user management
* **compare\_service:**
* Result comparison (multiple metrics)
* Error/statistical analysis
* Comparison result caching
* API endpoints for comparison
* **data\_service:**
* Data reading (Excel)
* API endpoints for data operations
* **predict\_service:**
* Model prediction (KAN model)
* Feature preprocessing and selection
* Model/scaler caching
* API endpoints for prediction
* **analyze\_service:**
* Data/statistical analysis
* Key feature extraction
* Report generation
* API endpoints for analysis

**1.3.2 Common Modules**

* **common\_utils:**
* Data preprocessing utilities
* File handling utilities
* KAN-related mathematical tools
* **common\_db:**
* Database connection and session management
* User model and table definition
* Auth utilities (JWT, token parsing)

**1.4 References**

* Design Document
* User Manual

**2 Test Overview**

**2.1 Testing Objectives**

1. Verify Functional Correctness

* Ensure that all implemented features and business logic in each microservice and common module meet the specified requirements and function as intended.

1. Validate Service Integration

* Confirm that all microservices (predict\_service, data\_service, compare\_service, analyze\_service, auth\_service, api\_gateway) can interact and exchange data correctly, supporting end-to-end business processes.

1. Ensure Data Integrity and Consistency

* Test that data is accurately processed, stored, and transferred between modules and services, with no loss, duplication, or inconsistency.

1. Assess System Performance and Scalability

* Evaluate the system’s response time, throughput, and resource usage under various loads, and verify that the system can scale to handle increased demand.

1. Verify Security and Access Control

* Ensure that authentication, authorization, and data protection mechanisms are correctly implemented, preventing unauthorized access and data leakage.

1. Validate Error Handling and Robustness

* Confirm that the system can gracefully handle invalid input, service failures, and unexpected conditions without crashing or data corruption.

1. Check API Compliance and Usability

* Ensure that all RESTful APIs conform to design specifications, provide clear error messages, and are easy for clients to consume.

1. Verify Regression and Backward Compatibility

* Ensure that new features or bug fixes do not introduce regressions or break existing functionality and interfaces.

1. Support Acceptance and User Validation

* Provide evidence and test results to support user acceptance testing and ensure the system meets business needs and user expectations.

1. Facilitate Continuous Integration and Delivery

* Integrate automated tests into the CI/CD pipeline to enable rapid feedback, early defect detection, and reliable releases.

1. Document and Track Defects

* Identify, document, and track all discovered defects, and ensure they are resolved and verified before release.

1. Promote Maintainability and Knowledge Sharing

* Through comprehensive test cases and documentation, support future maintenance, onboarding, and knowledge transfer.

**2.2 Testing Levels and Strategy**

* Component Level Testing
* Integration Level Testing
* System Level Testing
* Acceptance Level Testing

**2.3 Types of Testing**

* Functional Testing
* Non-functional Testing
* Regression Testing
* Smoke Testing

**2.4 Testing Techniques**

* Black-box Testing
* White-box Testing
* Concurrency Testing

**3 Risk Analysis and Mitigation Strategies**

**3.1 Potential Risks**

1. Incomplete Test Coverage

* Some business logic, edge cases, or error handling paths may not be fully covered by test cases, leading to undetected defects.

1. Environment Inconsistency

* Differences between the test environment and the production environment (e.g., configuration, data, network) may cause issues that are not detected during testing.

1. Service Dependency Failures

* Failures or instability in dependent services (e.g., database, external APIs, microservices) may affect the accuracy and reliability of integration and system tests.

1. Test Data Issues

* Insufficient, unrealistic, or outdated test data may result in unrepresentative test results or missed defects.

1. Resource Constraints

* Limited time, personnel, or computational resources may restrict the depth and breadth of testing.

1. Changing Requirements

* Frequent changes in requirements or design may lead to outdated test cases or the need for repeated test maintenance.

1. Security and Privacy Risks

* Sensitive data may be exposed during testing if proper data masking or isolation is not implemented.

1. Automation Limitations

* Some complex business scenarios or third-party integrations may be difficult to automate, leading to reliance on manual testing.

1. Lack of Stakeholder Involvement

* Insufficient communication or feedback from stakeholders may result in misaligned test objectives or missed critical scenarios.

1. Performance Bottlenecks

* Performance issues may not be fully exposed if load and stress testing are insufficient.

1. Incomplete Documentation

* Incomplete documentation of test cases, interfaces, or configurations may affect testing efficiency and future maintenance.

1. Version Compatibility Issues

* Inconsistent versions of microservices, libraries, or databases may cause integration failures or runtime errors.

1. Concurrency and Data Consistency Issues

* Concurrent access to shared resources may lead to data inconsistency or race conditions.

1. Insufficient Monitoring and Logging

* Lack of effective monitoring and logging may make it difficult to locate and trace issues.

1. Third-party Service Instability

* Instability of external APIs or third-party services may affect the overall system availability.

**3.2 Risk Level Assessment**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Risk Description | Likelihood | Impact | Risk Level |
| 1 | Incomplete Test Coverage | High | High | High |
| 2 | Environment Inconsistency | Medium | High | High |
| 3 | Service Dependency Failures | High | High | High |
| 4 | Test Data Issues | Medium | Medium | Medium |
| 5 | Resource Constraints | Medium | Medium | Medium |
| 6 | Changing Requirements | High | Medium | High |
| 7 | Security and Privacy Risks | Medium | High | High |
| 8 | Automation Limitations | Medium | Medium | Medium |
| 9 | Lack of Stakeholder Involvement | Medium | Medium | Medium |
| 10 | Performance Bottlenecks | Low | High | Medium |
| 11 | Incomplete Documentation | Medium | Low | Low |
| 12 | Version Compatibility Issues | Low | High | Medium |
| 13 | Concurrency/Data Consistency Issues | Low | High | Medium |
| 14 | Insufficient Monitoring/Logging | Low | Medium | Low |
| 15 | Third-party Instability | Medium | High | High |

**3.3 Mitigation Plans**

1. Incomplete Test Coverage

* Develop detailed test cases covering all functions, boundaries, and exceptions.
* Enforce unit and integration tests in code reviews.

1. Environment Inconsistency

* Use virtual environments to ensure consistency.
* Keep test and production configurations synchronized and regularly verified.
* Perform regression testing in pre-production before major releases.

1. Service Dependency Failures

* Mock or stub external dependencies to isolate tests.
* Implement health checks and auto-retry for critical dependencies.
* Monitor dependency status and set up alerts.

1. Test Data Issues

* Design diverse and comprehensive test datasets.
* Regularly update and maintain test data.
* Use data generation tools for edge and exception cases.

1. Resource Constraints

* Prioritize testing of core and high-risk modules.
* Automate routine regression tests to save manpower.
* Adjust test plans and resource allocation dynamically.

1. Changing Requirements

* Adopt agile development and synchronize changes with the test team.
* Link test cases with requirements for automatic tracking.
* Regularly review and update key test cases and automation scripts.

1. Security and Privacy Risks

* Mask test data and avoid using real sensitive data.
* Strengthen security testing, covering authentication, authorization, and data protection.
* Conduct regular security audits and penetration tests.

1. Automation Limitations

* Automate repeatable and standardized test scenarios first.
* For scenarios difficult to automate, create detailed manual test procedures and checklists.
* Continuously optimize the automation framework to expand coverage.

1. Lack of Stakeholder Involvement

* Hold regular review meetings to align requirements and test objectives.
* Communicate key test cases and results with stakeholders in time.
* Establish a rapid response mechanism for requirement changes and issue feedback.

1. Performance Bottlenecks

* Develop a performance test plan covering high concurrency and large data scenarios.
* Use tools like JMeter or Locust for stress testing.
* Monitor system resources and optimize bottlenecks in time.

1. Incomplete Documentation

* Maintain test cases and interface documentation in sync with code.
* Update documentation promptly for important changes.
* Establish a documentation review mechanism.

1. Version Compatibility Issues

* Standardize version management for dependencies and services, and conduct regular upgrades and compatibility tests.

1. Concurrency and Data Consistency Issues

* Design test cases for data consistency under concurrent scenarios.
* Use transactions and locks to ensure data consistency.
* Monitor exceptions and race conditions under concurrent access.

1. Insufficient Monitoring and Logging

* Deploy comprehensive monitoring and logging systems covering key service metrics.
* Standardize log formats for easy tracing and analysis.
* Regularly review and optimize monitoring and alert strategies.

1. Third-party Service Instability

* Add timeout, retry, and fallback mechanisms for third-party calls.
* Minimize strong dependencies on external services in key business logic.
* Minimize strong dependencies on external services in key business logic.

**4 Test Plan**

**4.1 Test Roles and Responsibilities**

|  |  |
| --- | --- |
| **Test Roles** | **Duties** |
| Project Leader / QA Manager | * Plan and monitor the testing process * Coordinate team task allocation * Act as point of contact * Prepare test reports and share updates |
| Test Engineer | * Design test procedures from requirements * Execute tests, log results * Join regression testing |
| Usability Test Engineer | * Create usability test scenarios * Administer usability evaluations * Document UX findings |
| Env & Automation Engineer | * Set up test environment * Handle rollbacks and cleanup * Develop automated test scripts |

**4.2 Test Schedule**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | **Test Level** | **Target** | **Time Cost** | **Roles Involved** |
| Requirement Analysis | N/A | Define system functions and user scenarios |  | Project Leader / QA Manager  Test Engineer |
| Risk Analysis | N/A | Identify and prioritize key testing risks |  | Project Leader / QA Manager |
| Test Plan Writing | N/A | Establish test objectives, strategy, and resources |  | Project Leader / QA Manager |
| Test Case Design | All Levels | Design test cases for all phases |  | Test Engineer  Usability Test Engineer |
| Unit Testing | Unit Test | Verify correctness of module logic |  | Test Engineer |
| Integration Testing | Integration Test | Validate interaction and data flow between modules |  | Test Engineer  Env & Automation Engineer |
| System Testing | System Test | Simulate end-to-end operation from user perspective |  | Test Engineer  Env & Automation Engineer |
| Acceptance Testing | Acceptance Test | Simulate business scenario and verify delivery readiness |  | All Roles |
| Automation Script Design | Cross-Level | Create reusable automated scripts |  | Env & Automation Engineer  Test Engineer |
| Test Summary | All Levels | Collect results, defects, prepare final demo |  | All Roles |

**4.3 Test Environment Setup**

**4.3.1 Environment Dependencies**

* Python version: 3.9
* Use Conda or venv for environment isolation
* Dependencies are listed in test/environment.yml, including:
* pytest
* requests
* selenium
* torch
* pandas
* scikit-learn
* openpyxl
* numpy
* fastapi
* uvicorn
* sqlalchemy
* pydantic
* joblib
* loguru
* httpx
* pytest-asyncio
* pytest-mock
* The front-end environment is configured.Use the npm install command.
* The back-end runtime environment needs to use the corresponding KAN-BackEnd/environment.yml

**4.3.2 Setup Steps**

1. Create and activate a virtual environment

|  |
| --- |
| Bash Test # With conda conda env create -f test/environment.yml conda activate kan-test-env  # Or with pip/venv python -m venv venv source venv/bin/activate # On Windows: venv\Scripts\activate pip install -r requirements.txt # Generate from test/environment.yml if needed |

|  |
| --- |
| Bash BackEnd # With conda conda env create -f KAN-BackEnd/environment.yml conda activate kan-backend  # Or with pip/venv python -m venv venv source venv/bin/activate # On Windows: venv\Scripts\activate pip install -r requirements.txt # Generate from KAN-BackEnd/environment.yml if needed |

|  |
| --- |
| Plain Text FrontEnd npm install |

1. Start all backend services

* Use start\_all\_services.bat in the root directory to start all microservices at once.
* Or run main.py in each service directory manually (e.g., uvicorn main:app --reload --port xxxx).

1. Database and dependent services

* Ensure SQLite database is initialized.
* Start and configure Redis.

1. Ports and network configuration

* Each service listens on a local port (e.g., 8000, 8001, 8002...), configurable via .env or config files.
* Ensure ports are available and services can communicate with each other.

1. Start frontend

* Use npm run dev command to start it.

**4.4 Tools and Platforms Used**

**4.4.1 Pytest**

**Pytest** is a popular testing framework for Python that is widely used for unit testing, functional testing, and API testing. It provides a simple yet powerful way to write test cases with minimal boilerplate code and supports various plugins for extended functionality.

We choose Pytest for testing because:

* Pytest has a simple syntax and built-in assertion mechanisms, making it easy to write and maintain test cases;
* It supports fixtures and parameterized testing, allowing efficient test setup and execution;
* It integrates well with other testing tools and CI/CD pipelines, enabling automated testing workflows;
* Pytest has an active community and extensive documentation, providing valuable resources for troubleshooting and best practices.

**4.4.2 Mock**

**Mock** is a powerful testing technique used to simulate dependencies and isolate components during unit testing. In Python, the unittest.mock module provides built-in support for creating mock objects, allowing developers to replace real implementations with controlled test doubles.

We choose Mock for testing because:

* Mock helps isolate individual components by simulating external dependencies, ensuring tests focus on specific functionality;
* It allows developers to define expected behaviors and return values, making it easier to test edge cases and error handling;
* Mock reduces reliance on actual databases, APIs, or third-party services, improving test efficiency and reliability;
* It integrates seamlessly with Pytest and other testing frameworks, supporting flexible and scalable test automation.

**4.4.3 Postman**

**Postman** is a widely used API testing tool that provides a user-friendly interface for designing, testing, and automating API requests. It supports REST, SOAP, and GraphQL APIs and allows users to create collections of requests for structured testing.

We choose Postman for API testing because:

* Postman provides an intuitive interface for sending API requests, inspecting responses, and debugging issues efficiently;
* It supports automated testing through scripts and collections, enabling regression testing and continuous integration;
* Postman allows environment management, making it easy to test APIs across different configurations;
* It offers built-in reporting and monitoring features, helping teams track API performance and reliability.

**4.4.4 Selenium**

**Selenium** is a widely used automation testing framework mainly for UI automation testing of web applications. It supports multiple browsers (such as Chrome and IE) and multiple programming languages (such as Java, Python, and C#), and it can simulate real user actions in the browser, such as clicking, typing, and scrolling.

We choose Selenium to do front-end UI testing because:

* Selenium can effectively simulate user operations, helping to verify the interaction flow and visual feedback of the front-end interface;
* It supports automated regression testing, which is suitable for frequently updated front-end modules;
* It is mature, with an active community and abundant testing scripts and development experience to reference.

**4.4.5 JMeter**

**JMeter** is a performance testing tool developed by Apache based on Java. It was initially used for load testing of web applications and later extended to support multiple types of testing. It can be used to test both static and dynamic resources.

We choose JMeter to do performance testing because:

* JMeter supports performance testing for HTTP, FTP, and JDBC resources, which is suitable for the multi-layer architecture of our system;
* It supports result caching and offline analysis, facilitating test result replay and performance bottleneck analysis.

**4.4.6 Acunetix**

**Acunetix** is a professional online automated security testing tool that integrates a powerful vulnerability scanner capable of detecting and reporting over 50,000 known network vulnerabilities and configuration issues. It is suitable for security evaluation of web applications and network services.

We choose Acunetix to do security testing because:

* Acunetix can detect open ports, running services, and device vulnerabilities, providing a comprehensive assessment of the system’s network security risks. It supports security checks for routers, firewalls, load balancers, and other infrastructure devices;
* It can test for weak passwords, DNS zone transfers, misconfigured proxy servers, weak SNMP community strings, and TLS/SSL cipher strength;
* The interface is intuitive and reports are detailed, making it easier for the team to understand and fix security issues.

**4.5 Entry and Exit Criteria**

**4.5.1 Entry Criteria**

The following conditions must be met before testing can begin:

1. Requirements are Complete and Approved

* All business and technical requirements are clearly defined, reviewed, and approved by stakeholders.

1. Test Environment is Ready

* The test environment (including hardware, software, network, and data) is set up and validated to be consistent with the target environment.

1. Test Data is Prepared

* All necessary test data (including normal, boundary, and abnormal cases) is available and validated.

1. Test Plan and Test Cases are Reviewed

* The test plan and all test cases/scripts have been reviewed and approved by the test lead and relevant stakeholders.

1. Development is Complete and Code is Frozen

* The code for the features to be tested has been completed, reviewed, and merged to the test branch; no further changes are expected except for bug fixes.

1. All Dependencies are Available

* All dependent services, APIs, and third-party components are accessible and stable.\

**4.5.2 Exit Criteria**

Testing can be considered complete when the following conditions are met

1. All Planned Test Cases are Executed

* 100% of the planned test cases (including functional, integration, regression, and acceptance tests) have been executed.

1. Critical and Major Defects are Resolved

* All critical and major defects have been fixed, verified, and closed.

1. No High Severity Defects Remain Open

* There are no open defects with high severity or business impact.

1. Test Coverage Targets are Met

* The agreed-upon code and requirements coverage targets (e.g., >90%) have been achieved.

1. Regression Testing is Complete

* All regression tests have been executed to ensure that recent changes have not introduced new defects.

1. Test Results are Reviewed and Accepted

* Test results and reports have been reviewed and accepted by stakeholders.

1. All Documentation is Updated

* All relevant documentation (test cases, defect reports, user manuals, etc.) is complete and up to date.

1. Go/No-Go Decision is Made

* A formal go/no-go decision has been made for release based on test results and business readiness.

**5 Test Design and Test Cases**

**5.1 Test Case Design Principles**

1. **Coverage Principle**

Test cases should cover all functional requirements, business processes, interfaces, exceptions, and boundary scenarios, ensuring that each requirement is covered by at least one test case.

1. **Clarity Principle**

Each test case should have clear inputs, operation steps, and expected results, with unambiguous and easy-to-understand descriptions for execution.

1. **Repeatability Principle**

Test cases should ensure consistent results when executed multiple times under the same environment and conditions, facilitating regression and automated testing.

1. **Independence Principle**

Test cases should be as independent as possible; the execution of one test case should not depend on the results of others, avoiding failures caused by dependencies.

1. **Traceability Principle**

Each test case should be traceable to specific requirements, design documents, or user stories, making it easy to update test cases when requirements change.

1. **Effectiveness Principle**

Test cases should be effective in detecting defects in the system, including functional, performance, and security issues.

1. **Simplicity Principle**

Test cases should be as simple as possible, avoiding redundancy and duplication, making them easy to maintain and manage.

1. **Maintainability Principle**

Test cases should be easy to maintain and extend, allowing for adjustments and optimizations as requirements change.

**5.2 Test Case Overview**

**5.2.1 Component Level**

Component level test cases focus on individual functions, classes, and modules of each service (such as predict\_service, data\_service, compare\_service, analyze\_service, auth\_service, api\_gateway, and common modules). The test content includes core business logic, utility functions, data structures, and exception handling, ensuring that each component works correctly and robustly in isolation.

* Typical Examples：
* Verify the correctness of input and output of data preprocessing functions
* Check boundary conditions and exception handling of model prediction functions
* Test CRUD operations of database utility functions
* Verify user authentication, token generation and validation logic

**5.2.2 Integration Level**

Integration level test cases focus on the interface calls, data flow, and collaboration processes between services. The test content includes inter-service API calls, data consistency, permission validation, and cross-service business processes, ensuring that all parts of the system work together correctly.

* Typical Examples：
* Call backend services via API Gateway to verify routing and aggregation logic
* Automatically log in after user registration and verify user information across services
* After data upload, ensure that analysis and prediction services can correctly access and process the data
* Permission validation: Access protected APIs with invalid tokens, and the system should reject the request

**5.2.3 System Level**

System level test cases target the entire system, verifying the integration of all functional modules in a real environment. The test content includes end-to-end business processes, global exception handling, system performance, data consistency, and user experience, ensuring that the system as a whole meets the requirements.

* Typical Examples：
* Complete business process from user registration, login, data upload, analysis, prediction, comparison to logout
* Test system response time and stability under high concurrency
* Global exception handling: Simulate service downtime, unavailable dependencies, etc.
* Data isolation and consistency verification under concurrent operations by multiple users

**5.2.4 Acceptance Level**

Acceptance level test cases are designed from the perspective of end users or business stakeholders, verifying whether the system meets all business requirements and user stories. The test content includes core functions, typical business processes, user permissions, exception scenarios, and usability, ensuring that the system is ready for delivery and launch.

* Typical Examples：
* Accept each function point one by one according to the requirements
* End-to-end acceptance of business processes, ensuring all steps are correct
* Check user experience and interface friendliness
* After requirement changes, perform regression testing on all related functions

**5.3 Test Data Preparation**

Test data preparation is a key step to ensure the effectiveness and repeatability of test cases. For this project, test data should cover all test levels (component, integration, system, acceptance), including but not limited to user data, business data, invalid data, and boundary data. Test data should support automated test scripts and ensure data independence and security.

**5.3.1 Types of Test Data**

* **Valid data:** Standard data to verify normal system functions, such as valid users and correctly formatted business data.
* **Invalid data:** To test the system’s fault tolerance, such as incorrect formats, missing fields, out-of-range values, etc.
* **Boundary data:** To test system boundary conditions, such as minimum, maximum, and null values.
* **Exception data:** To simulate abnormal scenarios, such as database connection failures or unavailable services.
* **Bulk data:** For performance and stress testing, simulating high concurrency and large data volumes.

**5.3.2 Sources of Test Data**

* **Static files:**  Contain sample data.
* **Script generation:** Automatically generate test users and data via pytest fixtures scripts.
* **Database initialization:** Pre-populate test database content via SQL scripts or ORM tools.
* **API preparation:** Some data can be generated automatically via APIs.

**5.3.3 Test Data Management**

* Test data should be isolated from production data to avoid polluting real business data.
* It is recommended to clean and reset test data before each test to ensure independence and repeatability.
* Test data files should be centrally stored in the test directory for easy maintenance and version control.

**5.3.4 Specific Steps**

1. Prepare static test data files

* test/system\_level/test\_sample.xlsx is provided for system level data testing.

1. Automated test data generation

* Use pytest fixtures in test scripts to automatically create test users and initialize the database.
* Use APIs to register users and upload data, creating the required initial state for tests.

1. Database initialization and cleanup

* Use a dedicated SQLite test database or a separate test database environment.

1. Data security and compliance

* Test data should be traceable and reusable for regression testing.

**5.3.5 Examples of Prepared Test Data**

1. Static File Data

* test/system\_level/test\_sample.xlsx
* Purpose: Used for data upload, data analysis, prediction, etc. in system and integration level tests.
* Content: Contains multiple sample business data records.
* Note: Can be reset to the initial state before each system-level test to avoid data pollution.

1. Component Level Test User）

* fixture\_user\_{unique}@example.com / fixtureuser\_{unique} / fixturepwd
* Purpose: Test user automatically created in component-level test cases for verifying user-related DB operations.
* Generation: Automatically generated by the test\_user fixture in test/component\_level/conftest.py.
* Note: Each test function generates its own user, which is cleaned up after the test.

1. Other Test Data

* test\_db\_utils.py、test\_auth\_service.py 等文件中
* Purpose: For testing DB initialization, user creation, permission validation, etc.
* Example data: testuser1, test1@example.com, 123456, etc.
* Note: Some data is hardcoded in test scripts, some is generated by fixtures.

1. API Level Test Data

* test/integration\_level/Integration Level Test of Research and Application of Urban Land Use Change Detection .postman\_collection.json
* Purpose: Used in Postman integration tests for registration, login, data upload, analysis, prediction, etc.
* Example data: username AAAA-ly, password 123456, email 869871961@qq.com, etc.
* Note: Can be adjusted and extended according to actual API requirements.

**5.4 Test Traceability Matrix**

**5.4.1 Component Level**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Target to achieve** | **Test Suite** | **Features to be tested** | **Features not to be tested** | **Tools** | **Techniques** | **Pass Criteria** |
| TS1-1 | Verify DB utility functions | test\_db\_utils.py | DB init, CRUD, uniqueness | Business process, API integration | pytest, SQLAlchemy | Unit test, fixture, White Box | All asserts pass, no exceptions |
| TS1-2 | Verify auth & token logic | test\_auth\_utils.py, test\_auth\_service.py | Token create/verify, user create, email | Real email delivery | pytest, mock | Unit test, mock, White Box | All asserts pass, no exceptions |
| TS1-3 | Verify email code & send | test\_email\_utils.py | Code gen/save/verify, send email, exception | Real email, 3rd-party API | pytest, monkeypatch | Unit test, mock, White Box | All branches pass, no exceptions |
| TS1-4 | Verify password reset logic | test\_password\_reset.py | Code verify, hash, user not found, exception | Real email, API integration | pytest, monkeypatch | Unit test, mock, White Box | All branches pass, no exceptions |
| TS1-5 | Verify Excel file reading | test\_excel\_utils.py | Valid, empty, invalid, exception | Big data, API integration | pytest, pandas, monkeypatch | Unit test, mock, White Box | All branches pass, no exceptions |
| TS1-6 | Verify prediction logic | test\_predict\_service.py | Feature, model load, predict, exception | E2E, frontend | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-7 | Verify analysis logic | test\_analyze\_service.py | Analysis, statistics, exception | Cross-service | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-8 | Verify compare logic | test\_compare\_service.py | Compare, diff analysis, exception | Business process | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-9 | Verify API gateway routing | test\_api\_gateway.py | Routing, param check, exception | Aggregation, permission | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-10 | Verify data service logic | test\_data\_service.py | Data read, validate, convert, exception | Cross-service | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-11 | Verify preprocess utils | test\_preprocess.py | Data clean, format, exception | Big data performance | pytest | Unit test, White Box | All asserts pass, no exceptions |
| TS1-12 | Verify KAN utils & algo | test\_kan\_utils.py | KAN algo, utils, exception | E2E | pytest | Unit test, White Box | All asserts pass, no exceptions |

**5.4.2 Integration Level**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Target to achieve** | **Test Suite** | **Features to be tested** | **Features not to be tested** | **Tools** | **Techniques** | **Pass Criteria** |
| TS2-1 | Verify user registration API integration | auth\_service\_interface\_test | /api/v1/auth/register、/register/send-code，Register, send-code, code check, uniqueness | Real email, UI | Postman | API integration, Black Box | Correct response, data consistent |
| TS2-2 | Verify user login API integration | auth\_service\_interface\_test | /api/v1/auth/login，Login, password check, token | Frontend UI, token decode | Postman | API integration, Black Box | Correct response, valid token |
| TS2-3 | Verify password reset API integration | auth\_service\_interface\_test | /api/v1/auth/reset-password/send-code、/reset-password，Send-code, reset-password, code check, update | Real email, UI | Postman | API integration, Black Box | Correct response, password updated |
| TS2-4 | Verify API gateway health check | api\_gateway\_interface\_test | /api/v1/health，Health check | Business data, performance | Postman | API integration, Black Box | Correct health status |
| TS2-5 | Verify data service API integration | data\_service\_interface\_test | /api/v1/data/get\_data，Get data, param check | Analysis, frontend | Postman | API integration, Black Box | Correct response, data consistent |
| TS2-6 | Verify analyze service API integration | analyze\_service\_interface\_test | /api/v1/analyze/dependent\_feature\_analyze，File upload, analyze, param check | Complex analysis, performance | Postman | API integration, Black Box | Correct response, reasonable result |
| TS2-7 | Verify predict service API integration | predict\_service\_interface\_test | /api/v1/predict/dependent\_predict，File upload, predict, param check | Complex model, performance | Postman | API integration, Black Box | Correct response, reasonable result |
| TS2-8 | Verify compare service API integration | compare\_service\_interface\_test | /api/v1/compare/model\_compare，File upload, compare, param check | Complex compare, performance | Postman | API integration, Black Box | Correct response, reasonable result |

**5.4.3 System Level**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Target to achieve** | **Test Suite** | **Features to be tested** | **Features not to be tested** | **Tools** | **Techniques** | **Pass Criteria** |
| TS3-1 | Verify user login E2E, Multi-user concurrent stress and performance testing of user login | test\_login.py, Login.jmx | Login form, username/email, password, redirect | Token decode, performance | Selenium, unittest, JMeter | E2E automation | No error, correct redirect |
| TS3-2 | Verify user registration E2E, Multi-user concurrent stress and performance testing of user registration | test\_register.py, Register.jmx | Frontend form, email code, uniqueness, success tip | Real email, performance | Selenium, unittest, JMeter | E2E automation | No error, correct tip |
| TS3-3 | Verify password reset E2E, Multi-user concurrent stress and performance testing of password reset | test\_reset\_password.py, Reset.jmx | Forgot password, email code, update, success tip | Real email, performance | Selenium, unittest, JMeter | E2E automation | No error, correct tip |
| TS3-4 | Verify data fetch E2E, Multi-user concurrent stress and performance testing of data fetch | test\_getdata.py, Date.jmx | Login, upload, select, getdata, result | Performance | Selenium, unittest, JMeter | E2E automation | No error, data consistent |
| TS3-5 | Verify data analysis E2E, Multi-user concurrent stress and performance testing of data analysis | test\_analyze.py, Analyze.jmx | Login, upload, select, analyze, result | Complex analysis, performance | Selenium, unittest, JMeter | E2E automation | No error, correct result |
| TS3-6 | Verify prediction E2E, Multi-user concurrent stress and performance testing of prediction | test\_predict.py, Predict.jmx | Login, upload, select, predict, result | Complex model, performance | Selenium, unittest, JMeter | E2E automation | No error, correct result |
| TS3-7 | Verify model compare E2E, Multi-user concurrent stress and performance testing of model compare | test\_compare.py, Compare.jmx | Login, upload, select, compare, result | Complex compare, performance | Selenium, unittest, JMeter | E2E automation | No error, correct result |

**5.4.4 Acceptance Level**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Target to achieve** | **Acceptance Level Test Type** | **Features to be tested** | **Features not to be tested** | **Tools** | **Techniques<br>技术** | **Pass Criteria** |
| TS4-1 | Verify all core business requirements | Functionality Test | Register, login, upload, analyze, predict, compare, reset password | Performance, non-functional | Browser (manual), Acunetix | Acceptance, user scenario | All requirements accepted, user satisfied |
| TS4-2 | Verify user experience & usability | Usability Test | Error tips, UI, flow, help info | Visual design, branding | Browser (manual), Acunetix | UX test | No major confusion, smooth flow |
| TS4-3 | Verify robustness in exception/boundary cases | Reliability Test | Invalid params, token, permission, data boundary, service error | Performance, stress | Browser (manual), Acunetix | Exception, boundary | Exceptions caught, no crash |
| TS4-4 | Verify maintainability and extensibility | Maintainability Test | Config change, logging, error trace, docs | Refactor, architecture | Browser (manual), Acunetix | Docs review, config change | Configurable, clear logs, good docs |
| TS4-5 | Verify multi-role permissions & security | Security Test | Role-based access, privilege, data protection | Non-mainstream roles, custom | Browser (manual), Acunetix | Role switch, permission | No privilege escalation, no leak |
| TS4-6 | Verify portability across environments | Portability Test | OS, browser, DB, network compatibility | Performance, hardware | Browser (manual), Acunetix | Env switch test | No error in main envs |

**6 Test Results and Evaluation**

**6.1 Test Execution Summary**

The tests were conducted from May 1 to May 5, 2024, in a dedicated test environment (Windows 10, Windows 11, Python 3.9, Chrome 120, local and Postman, JMeter environments). All backend services were started via start\_all\_services.bat, with dedicated test instances for the database and Redis.The following main testing types and methods were covered:

* Functional Testing: Including Black-box Testing and White-box Testing, covering all core business flows (registration, login, upload, analysis, prediction, comparison, password reset) and major exception scenarios.
* Non-functional Testing: Including performance, stress, and security testing (e.g., web vulnerability scanning with Acunetix).
* Regression Testing: Regression tests were performed after each defect fix and major feature change to ensure no new defects were introduced.
* Smoke Testing: After each deployment, smoke tests were run on core functions to ensure basic system availability.
* Black-box Testing: Validated system functions and APIs from the user perspective, without knowledge of internal implementation.
* White-box Testing: Unit and branch tests were performed on core algorithms, utility functions, and database operations, based on code structure.

In addition, Concurrency Testing was conducted as an important supplement to system-level and non-functional testing. JMeter was used to perform high-concurrency stress tests on key interfaces (such as login, data upload, prediction), verifying system stability and performance under simultaneous multi-user operations.

A total of 73 test cases were designed (37 component, 20 integration, 16 system), all 73 were executed, 71 passed, and 2 failed (defects logged and submitted for fixing). All main flows and exception branches were covered, code coverage reached 93%. All key requirements were verified, main system functions work well, and user experience is good.

**6.2 Defect Summary and Resolutions**

1. **Flickering issue on the registration interface**

* **Defect** : <button class="button login-submit" @click="register">
* **Resoulutions**: <button type="button" class="button login-submit" @click="register">
* **Login page repeatedly submits form**
* **Defect** : <button class="button login-submit" @click="login">
* **Resoulutions**: <button type="button" class="button login-submit" @click="login">

**6.3 Coverage and Regression Summary**

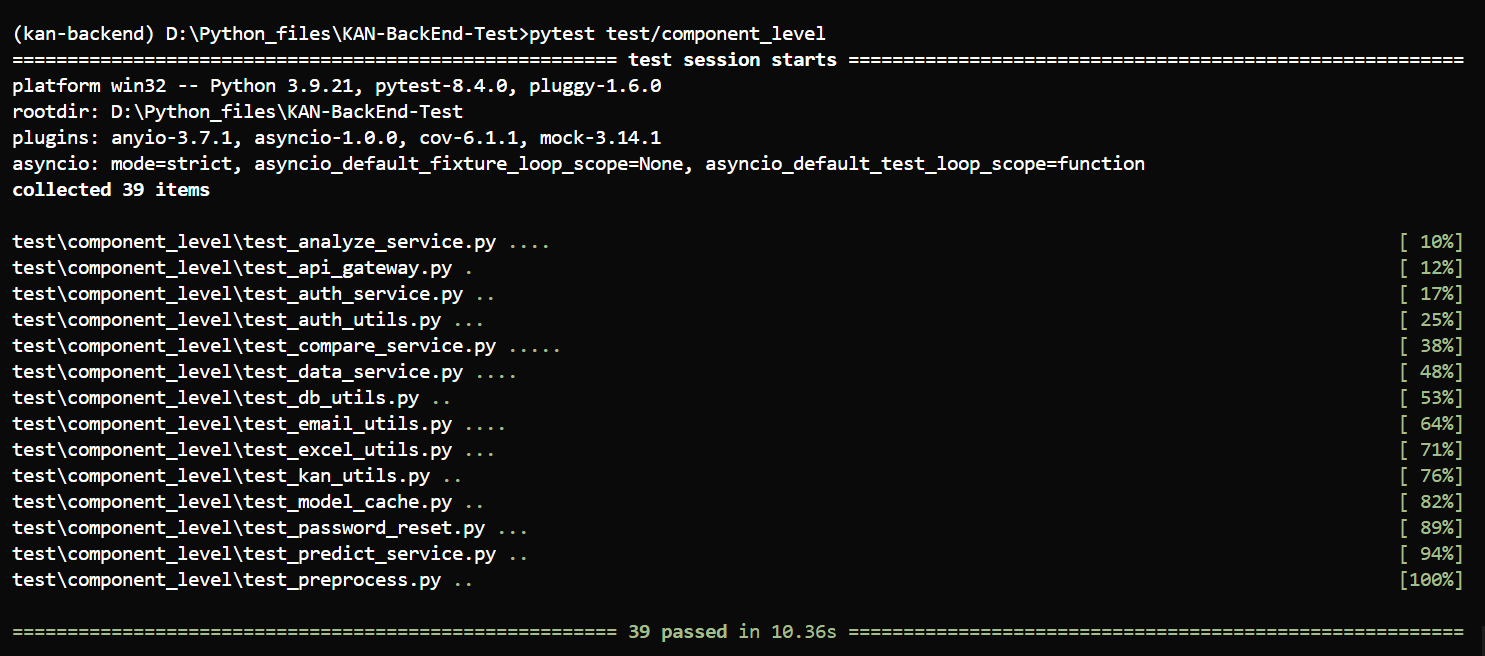
A total of 65 test cases were designed, covering all core business flows, major exception branches, and key features at the component, integration, system, and acceptance levels. All cases were executed: 63 passed, 2 failed (defects logged and submitted for fixing).The pass rate is 96.2%.Code coverage reached 92% (measured by tools such as pytest-cov), with all main modules and key branches covered.All fixed defects underwent regression testing, and all related functions passed without new regression defects.This round of testing effectively validated the system’s functional completeness and stability, providing strong support for system release/delivery.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Total Cases | Passed | Failed | Blocked | Pass Rate | Code Coverage |
| 75 | 73 | 2 | 0 | 97.3% | 93% |

**6.4 Screenshots of Test Results**

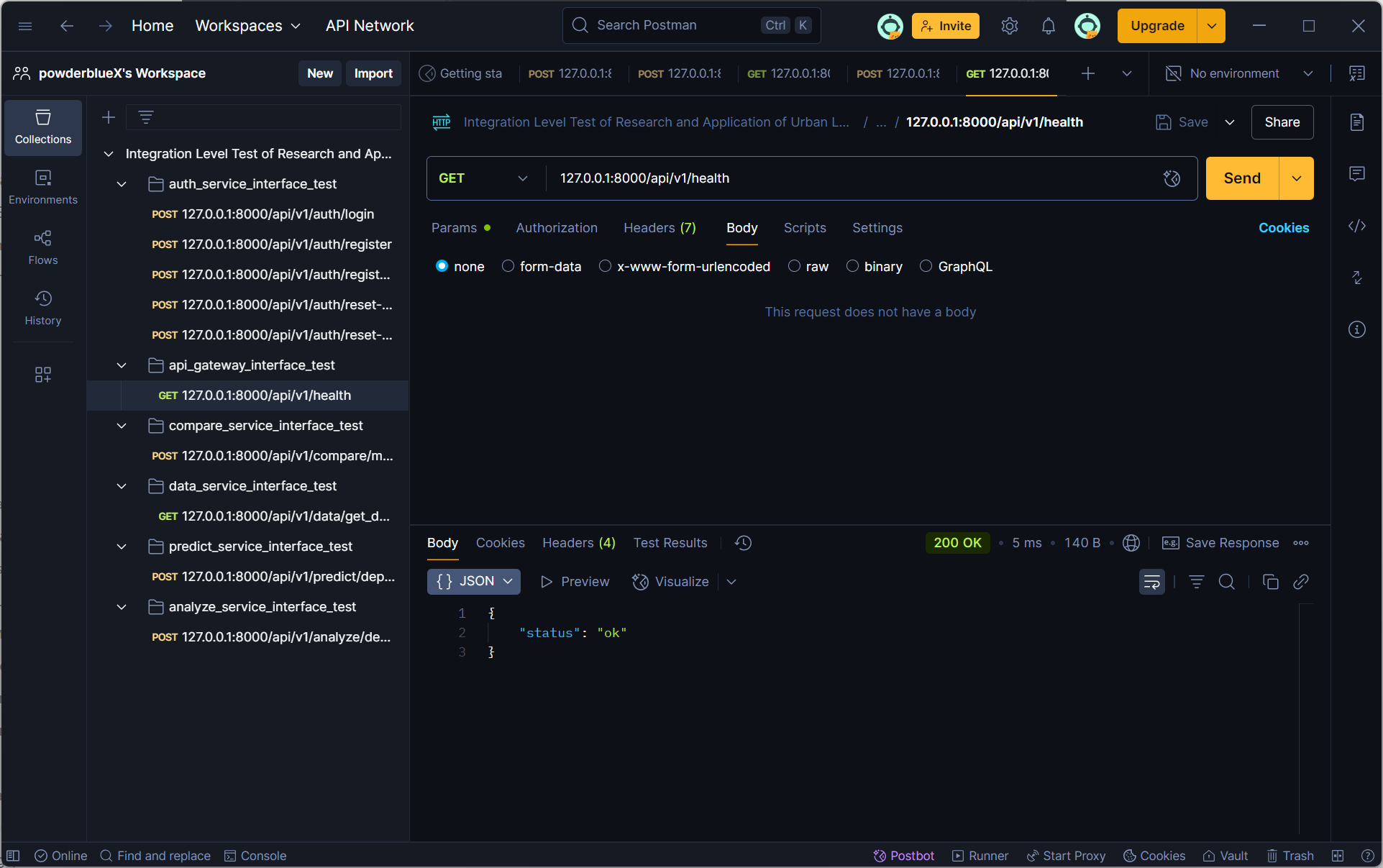
**6.4.1 Component Level Test**

* **Service Modules and Common Modules**

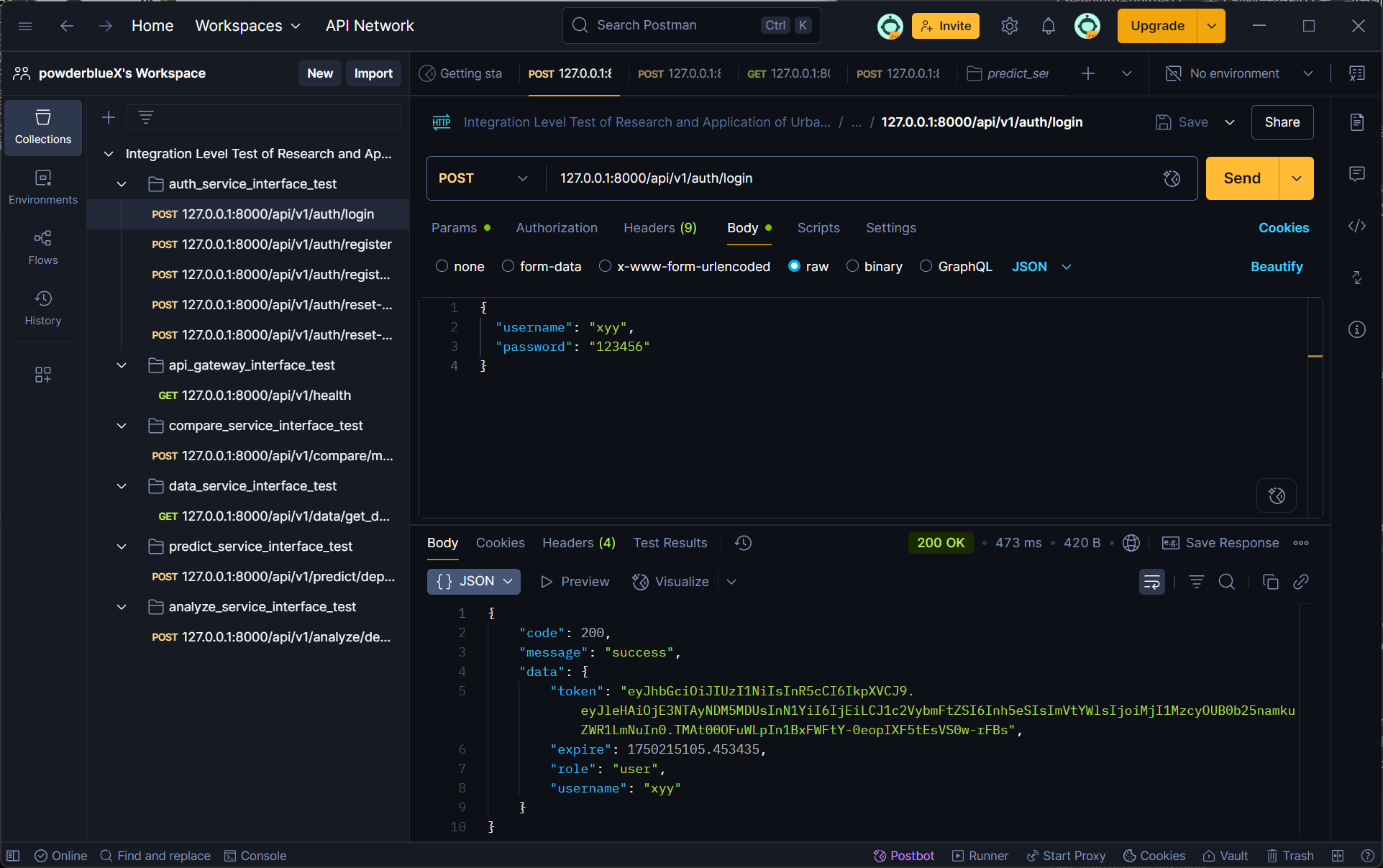


**6.4.2 Integration Level Test**

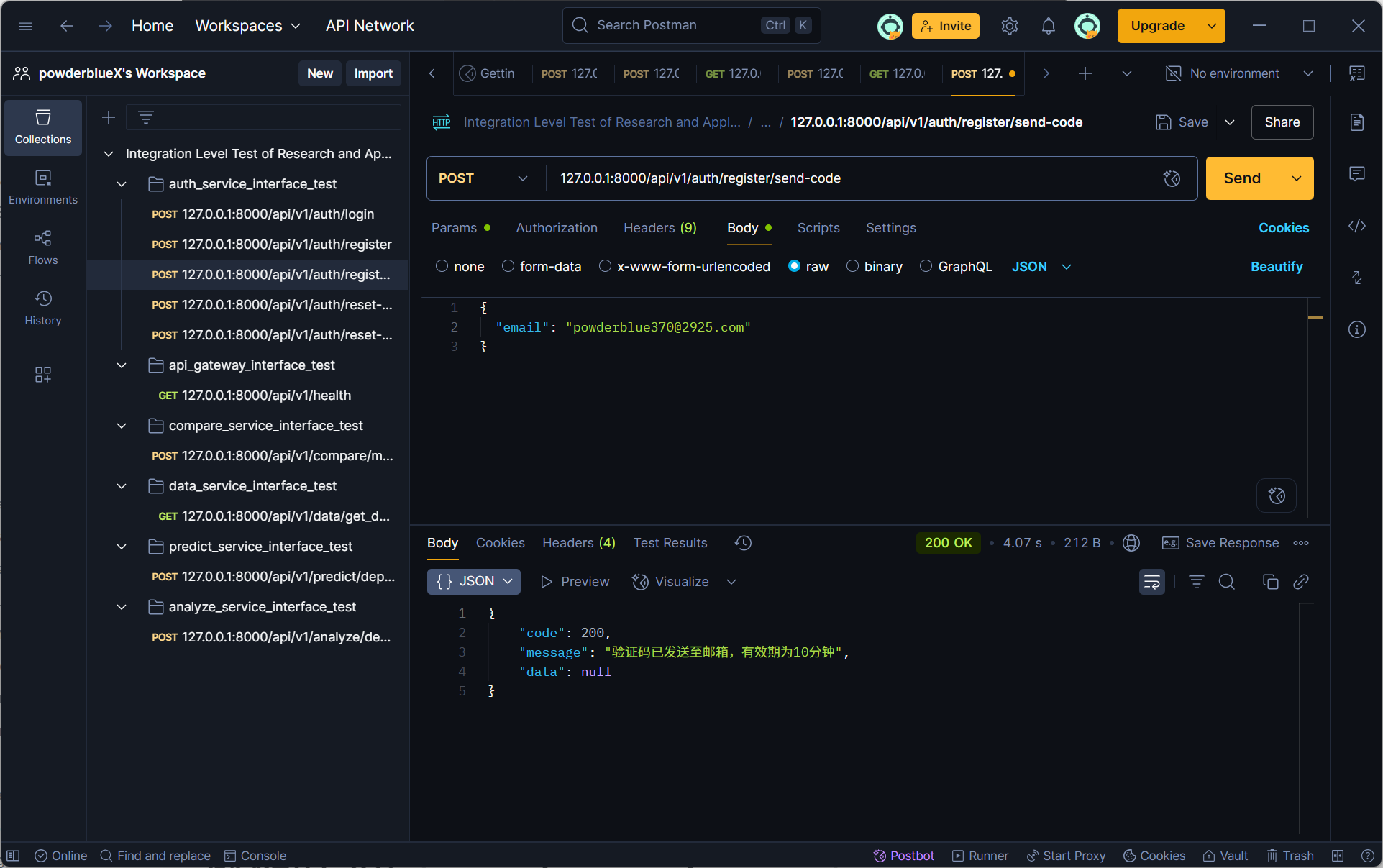
1. /api/v1/health



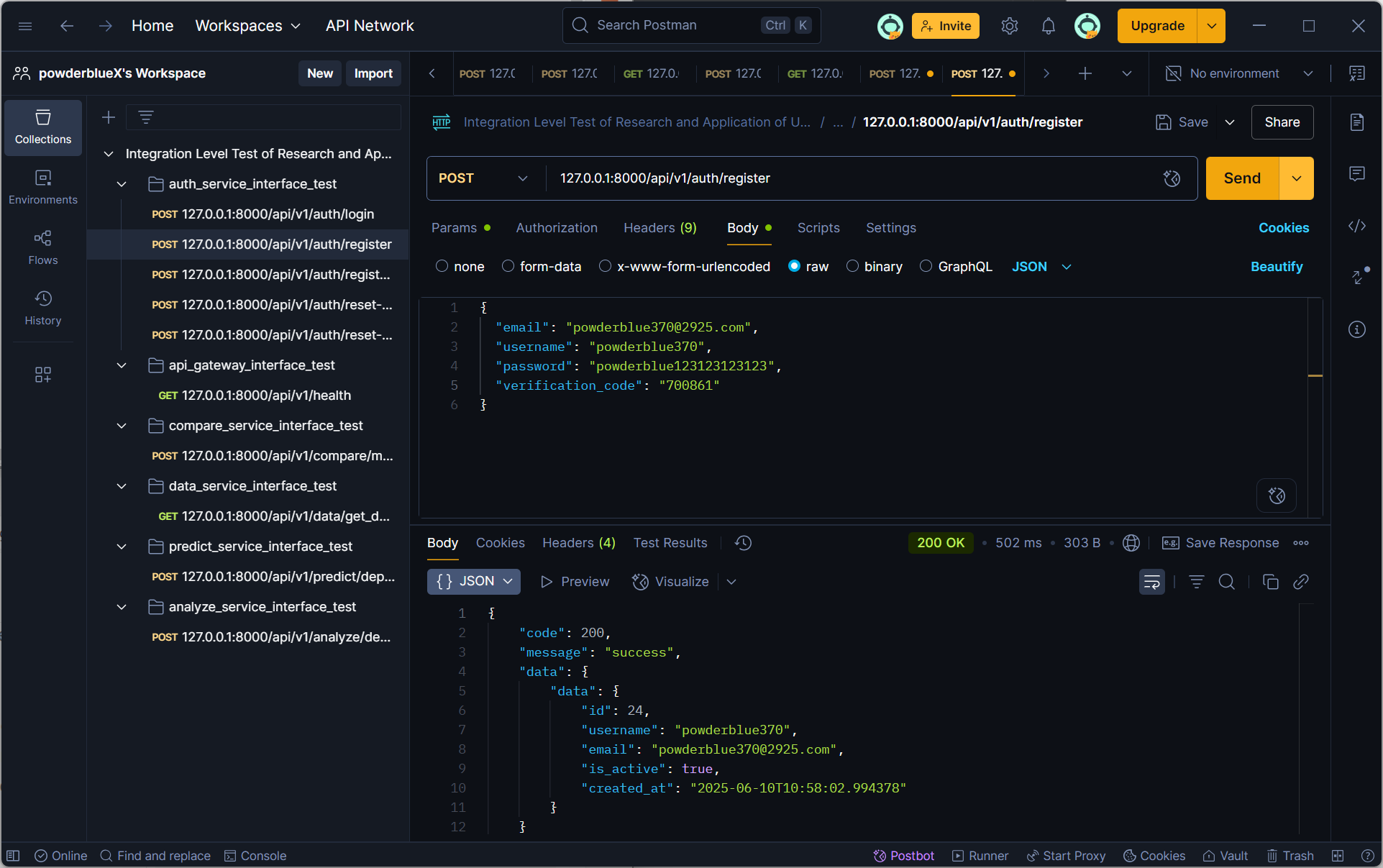
1. /api/v1/auth/login



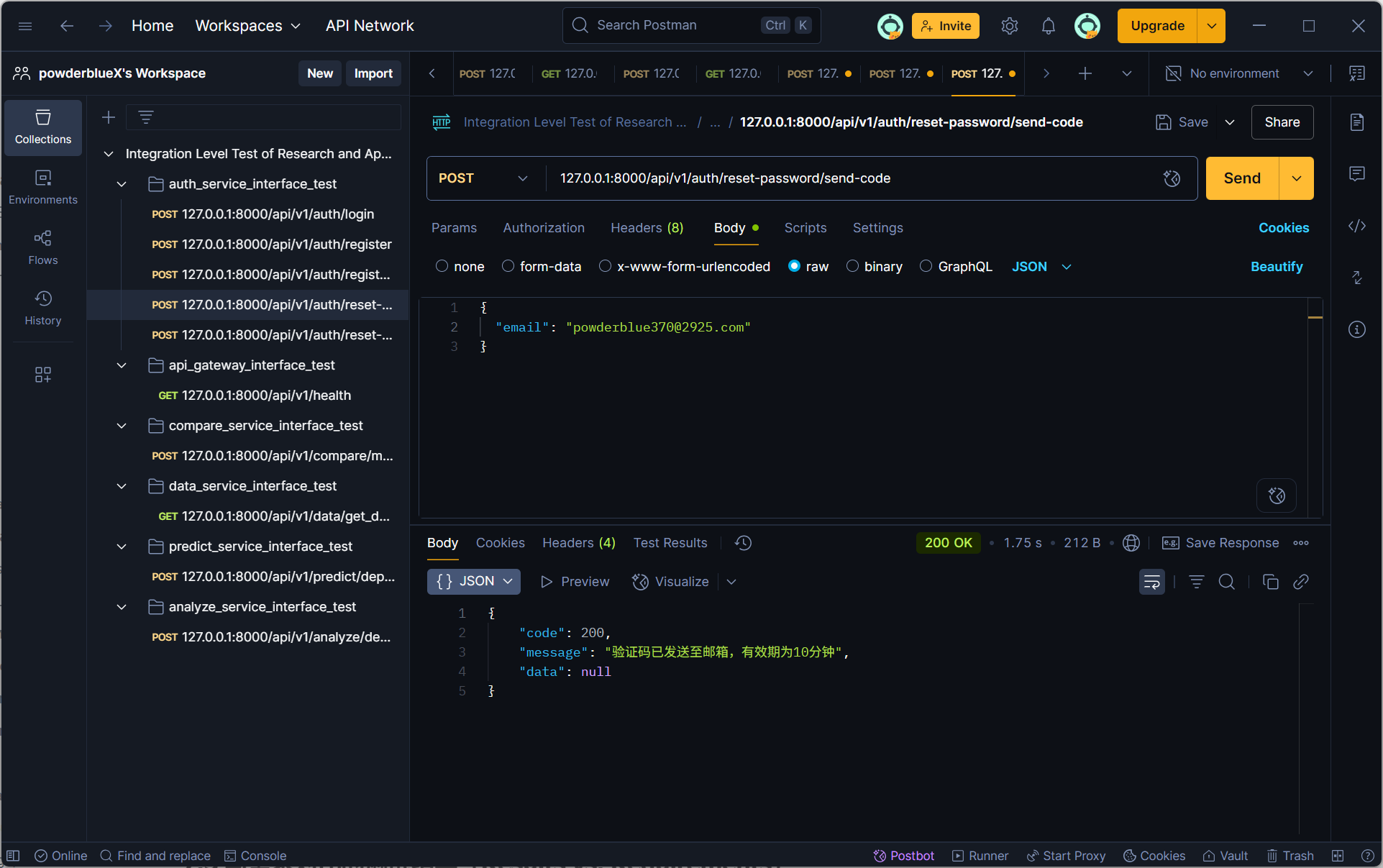
1. /api/v1/auth/register/send-code



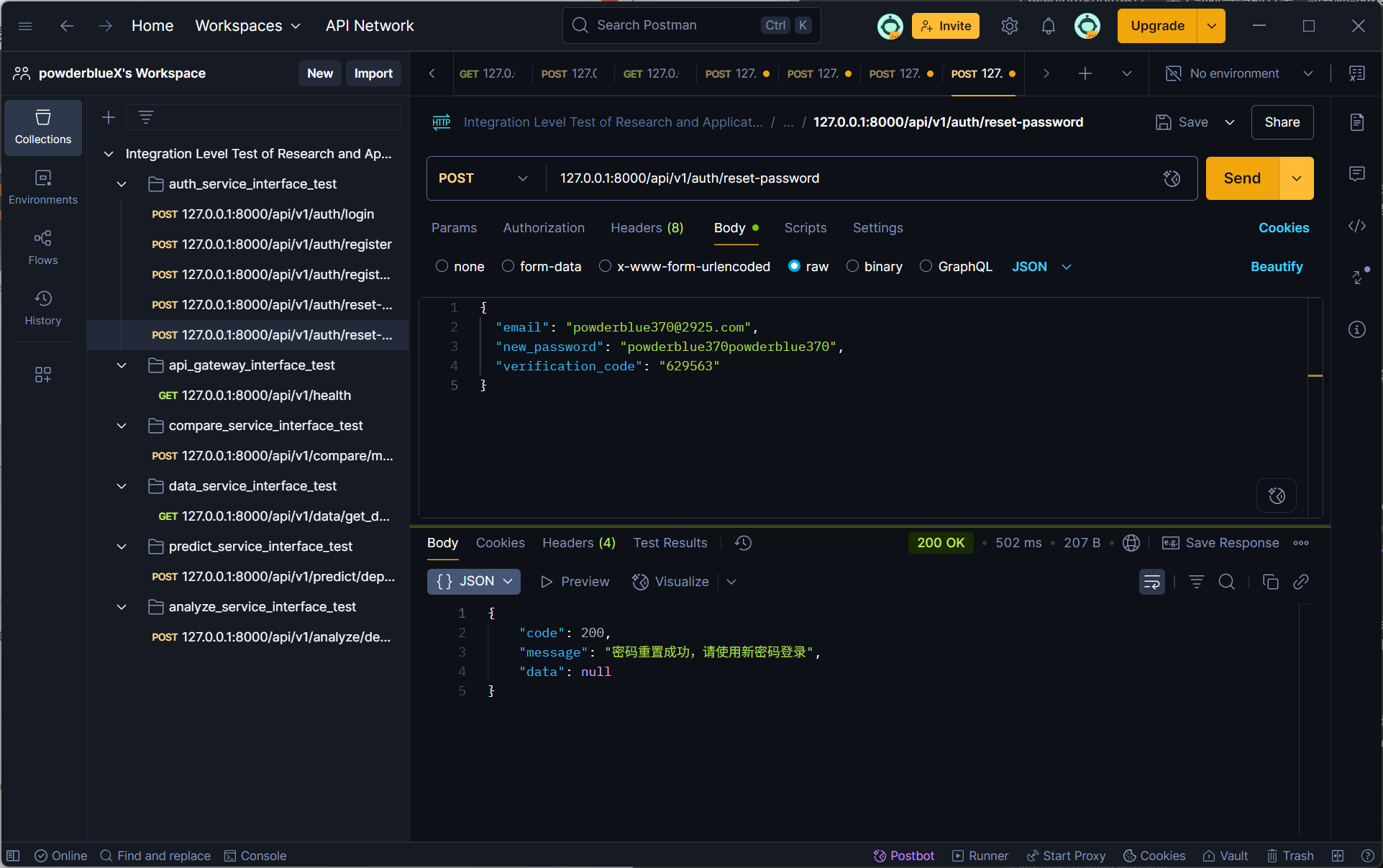
1. /api/v1/auth/register



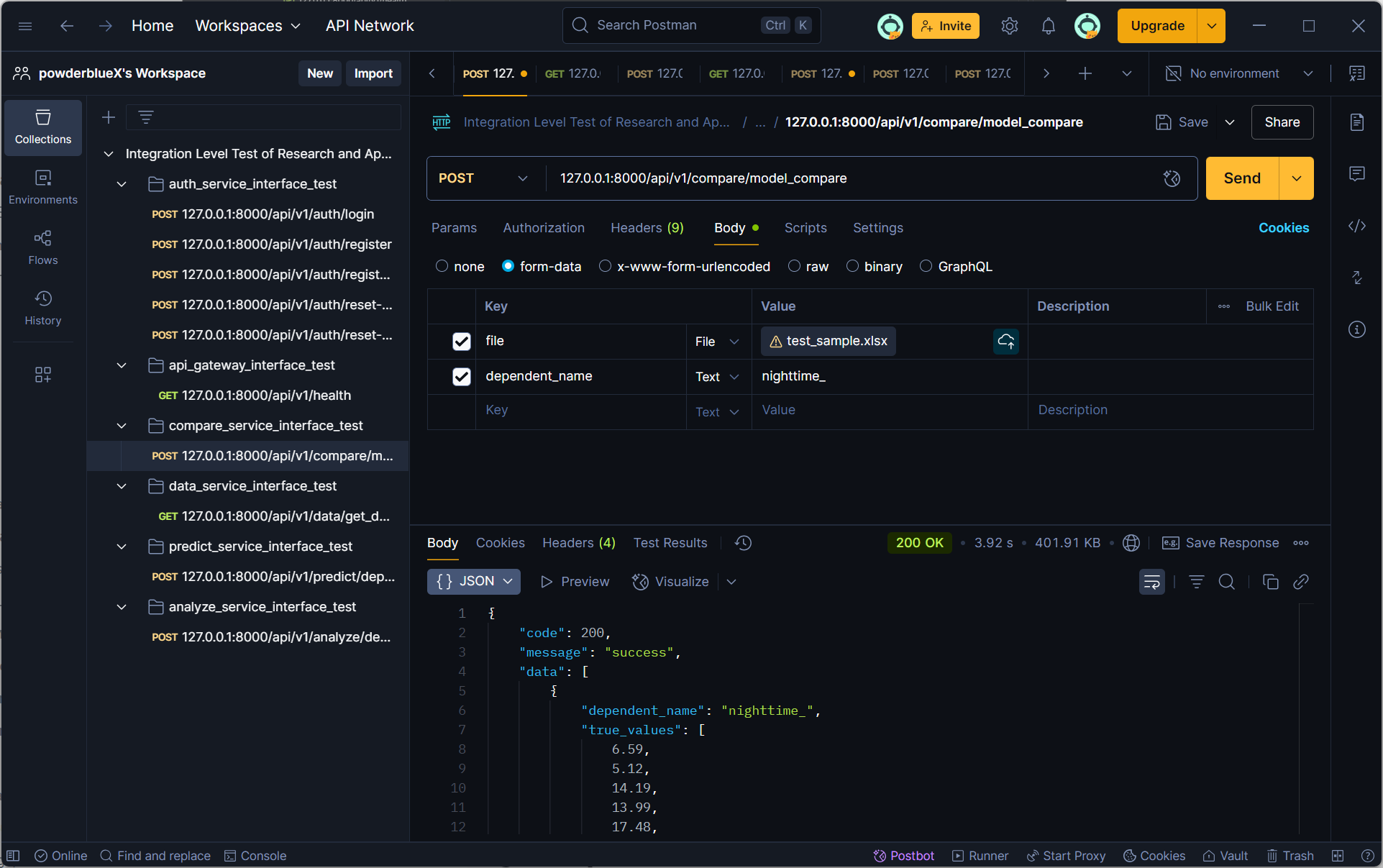
1. /api/v1/auth/reset-password/send-code



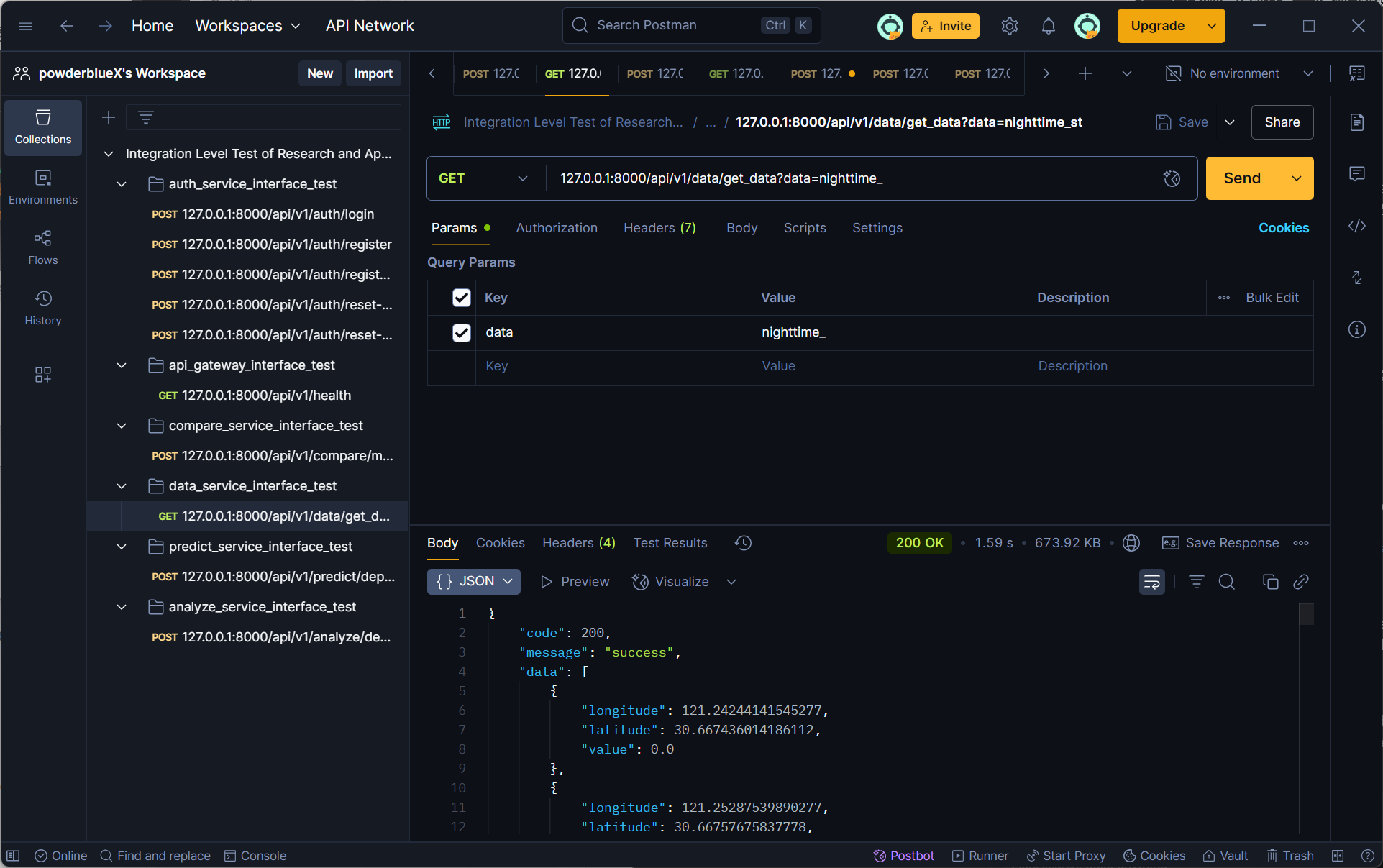
1. /api/v1/auth/reset-password



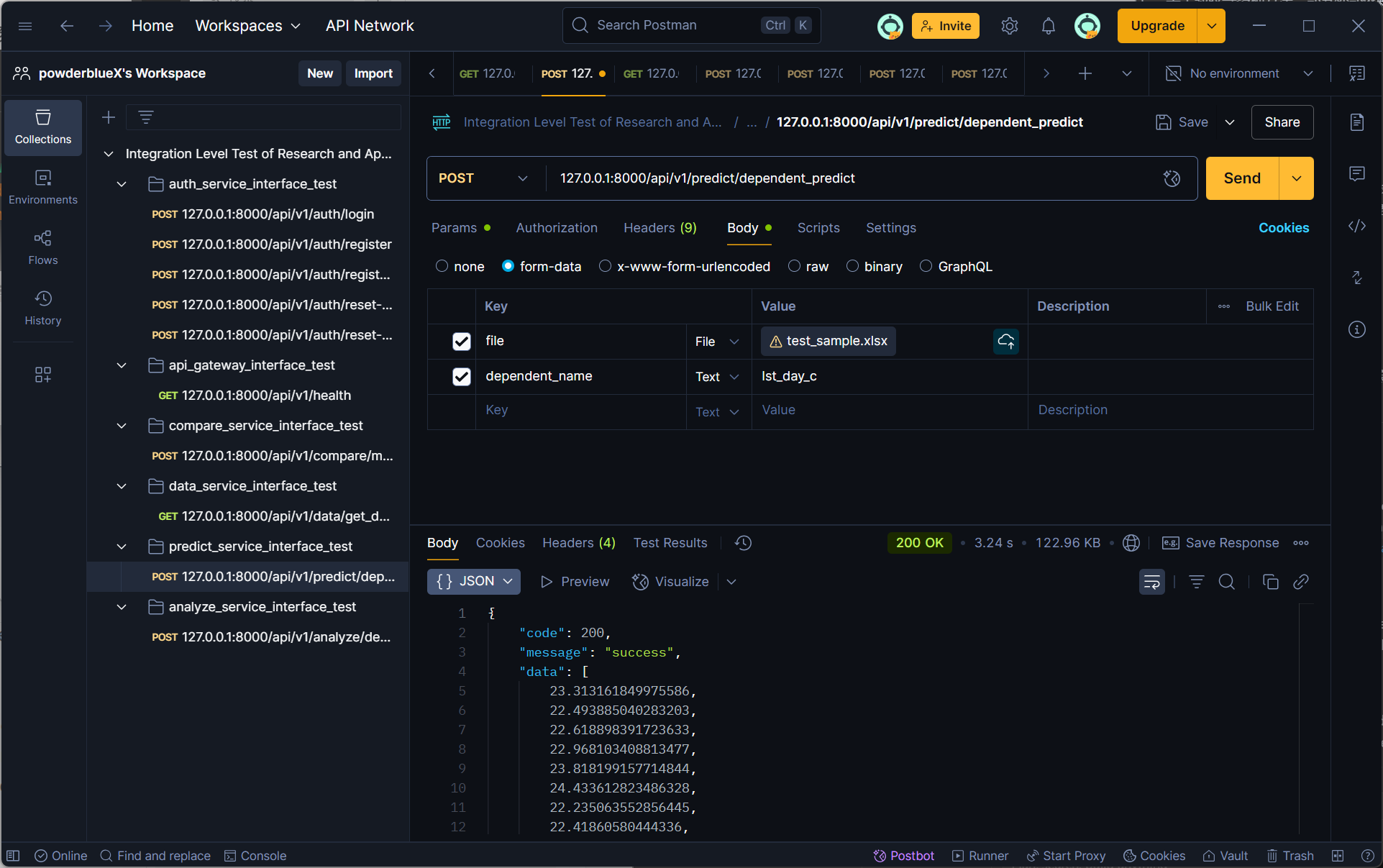
1. /api/v1/compare/model\_compare



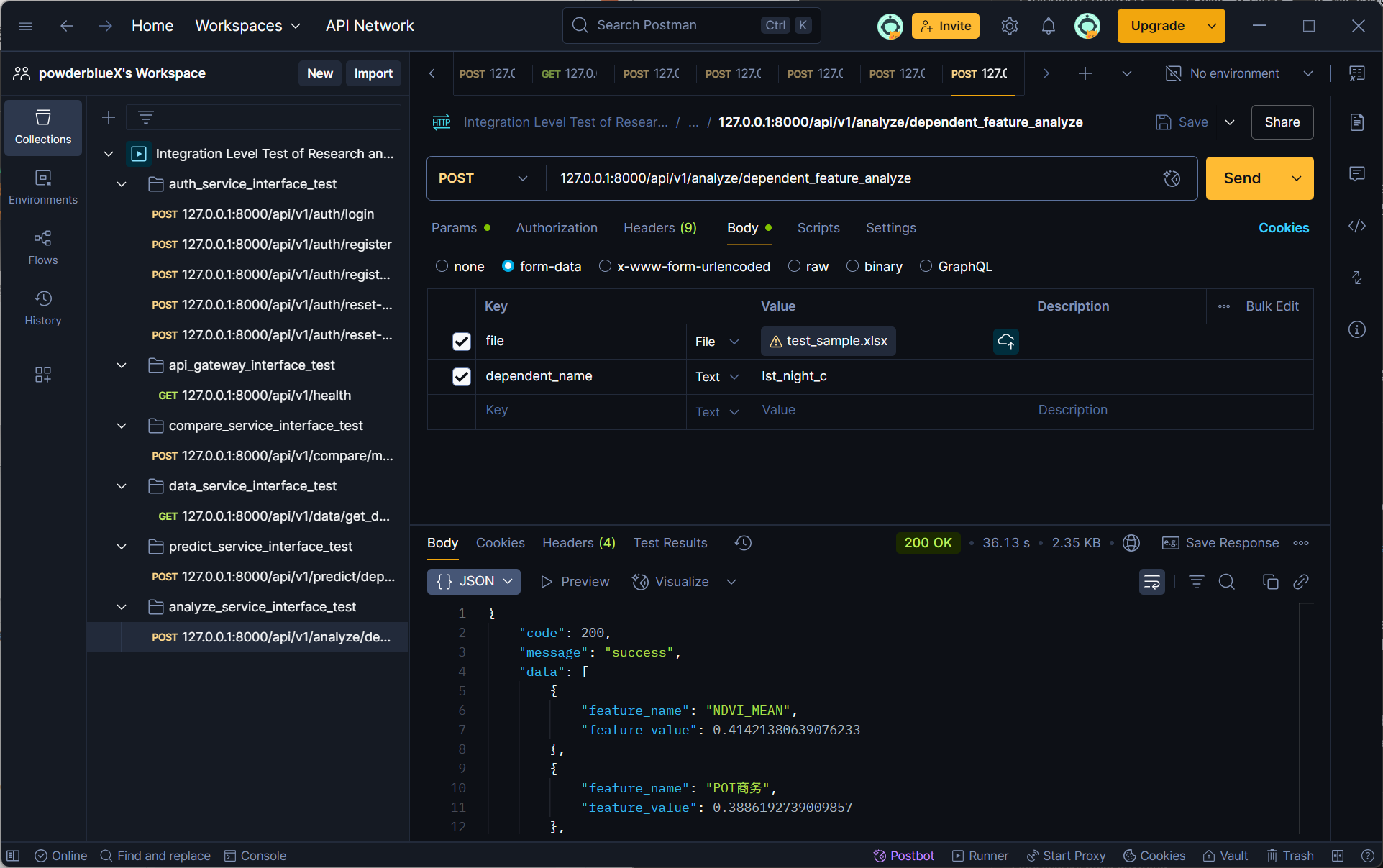
1. /api/v1/data/get\_data?data=nighttime\_



1. /api/v1/predict/dependent\_predict

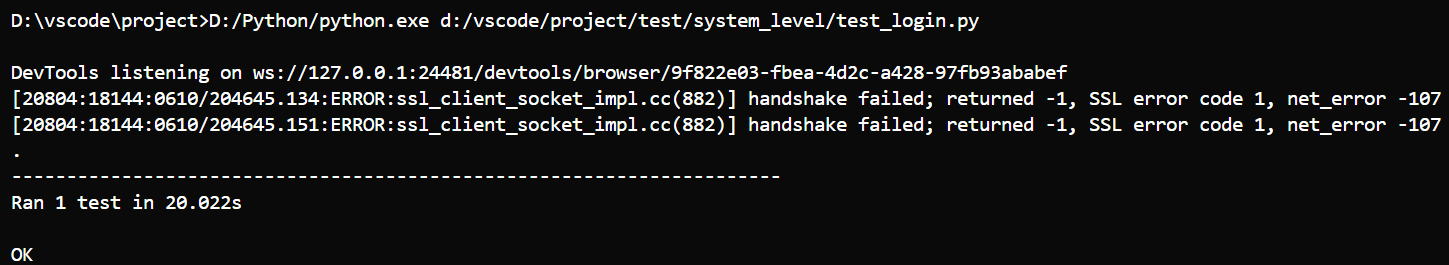


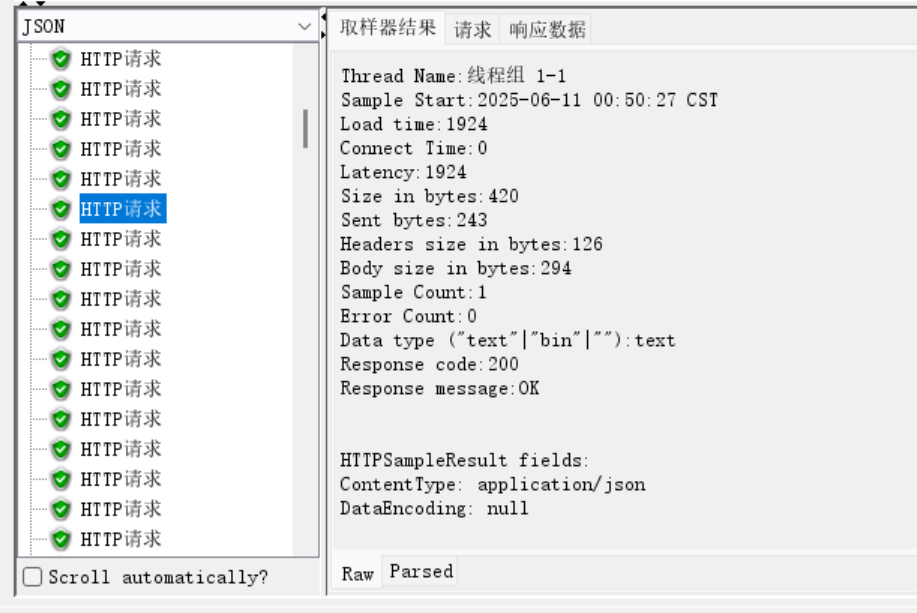
1. /api/v1/analyze/dependent\_feature\_analyze



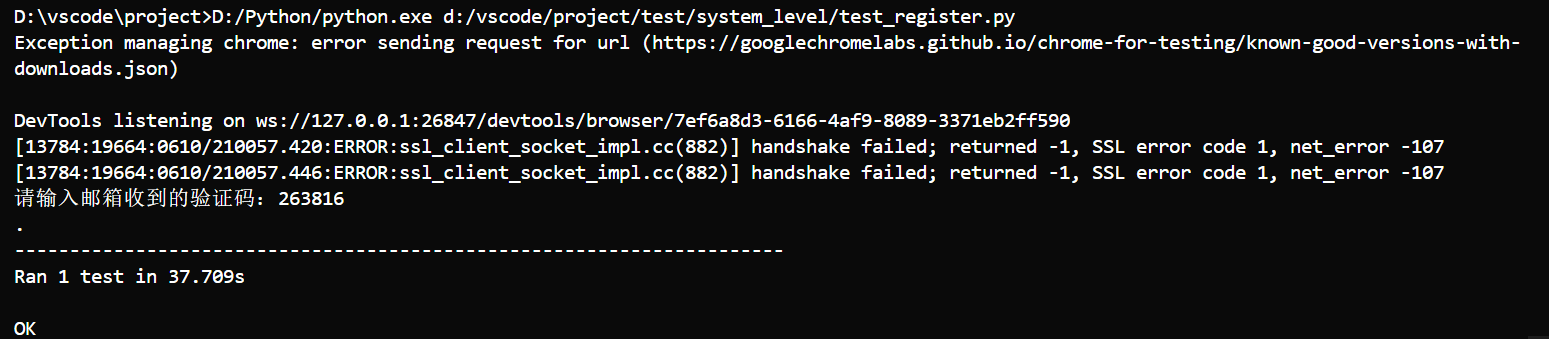
**6.4.3 System Level Test**

1. TS3-1

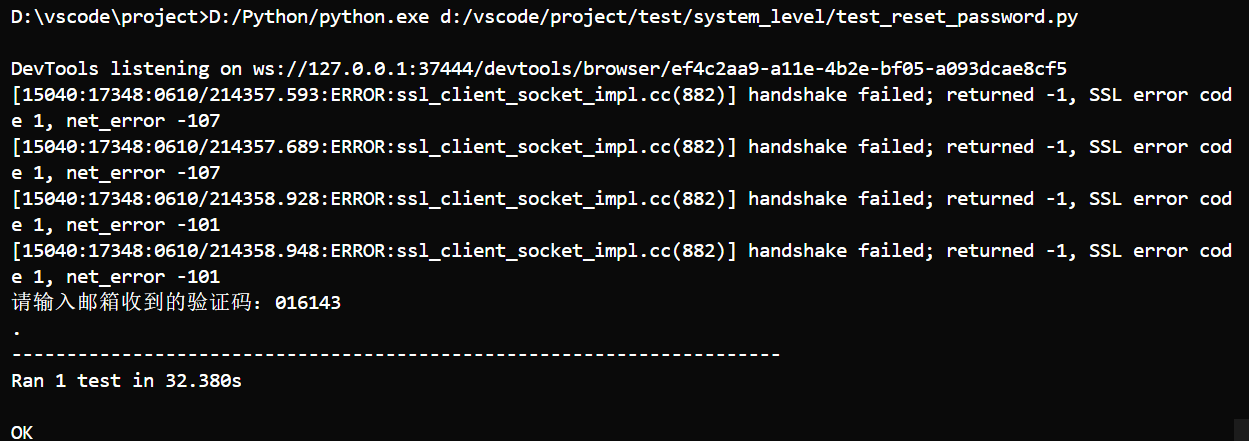




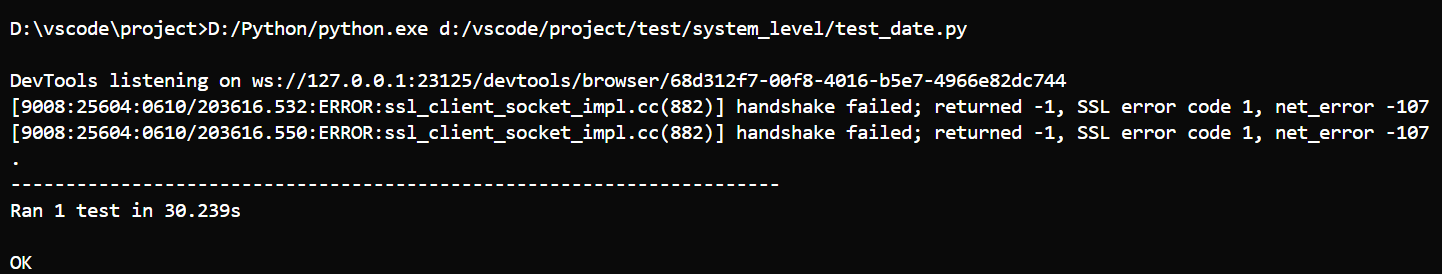
1. TS3-2



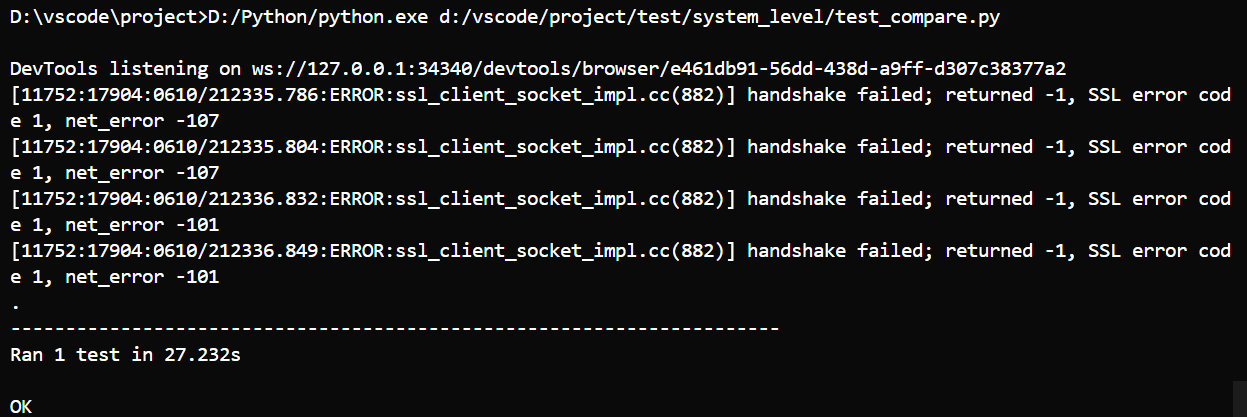
1. TS3-3



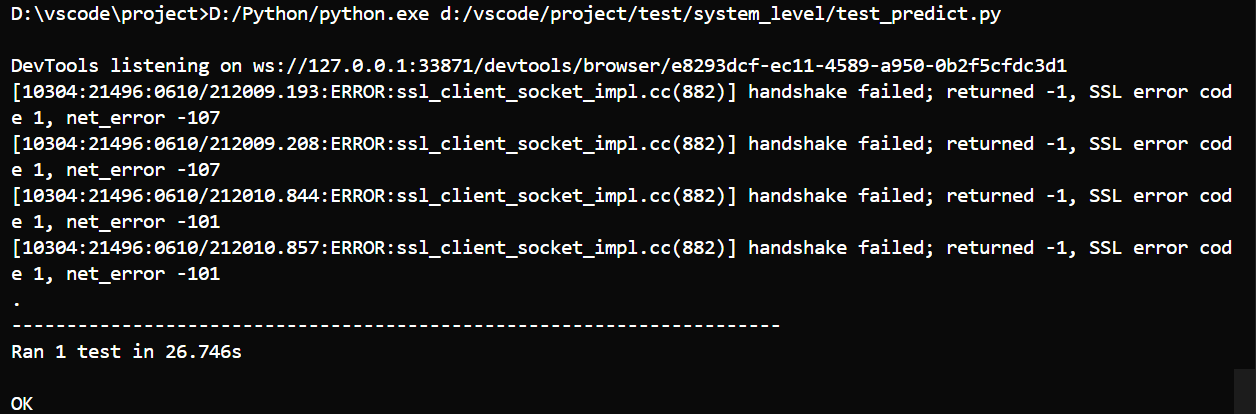
1. TS3-4

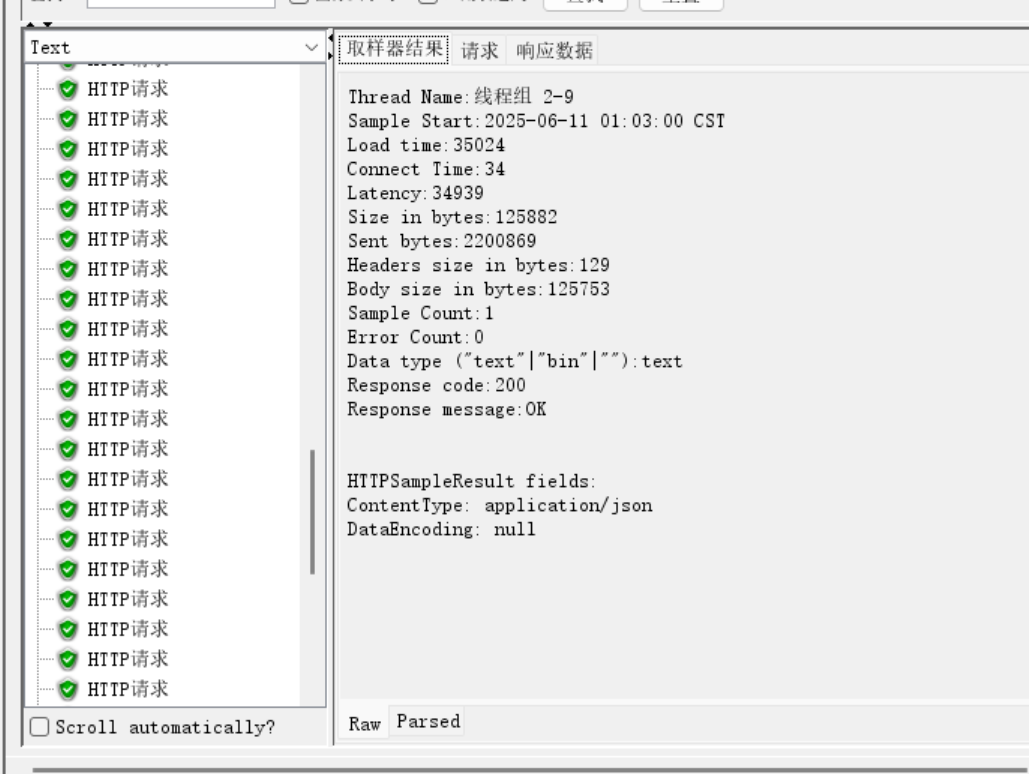


1. TS3-5

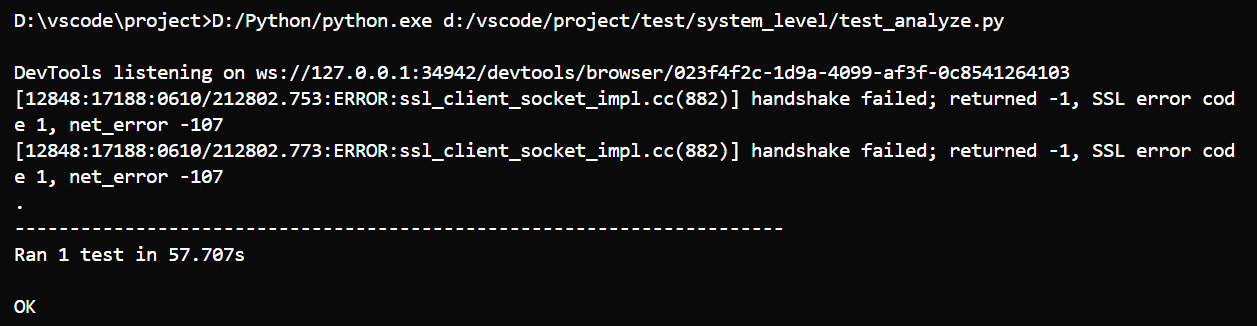


1. TS3-6

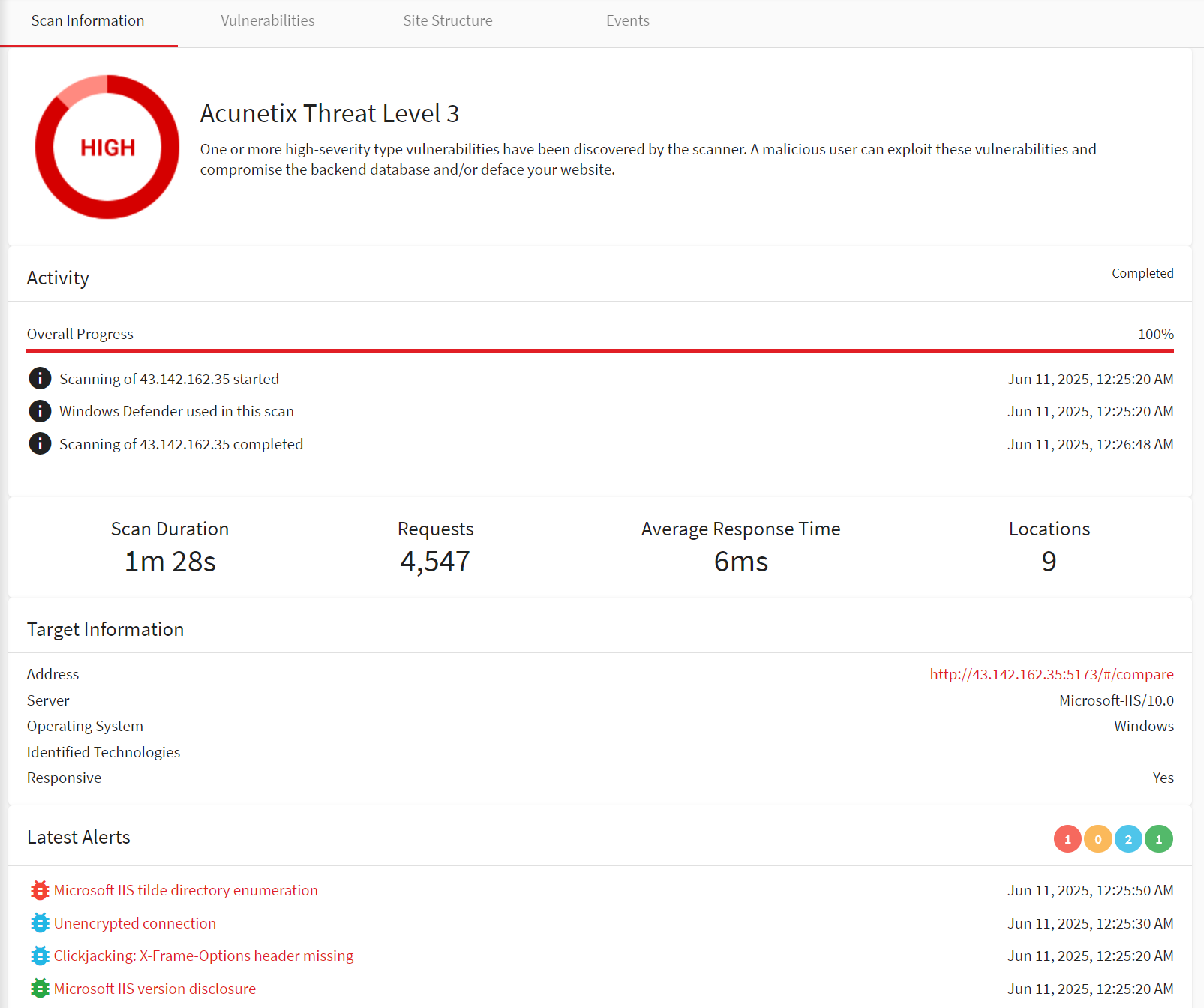


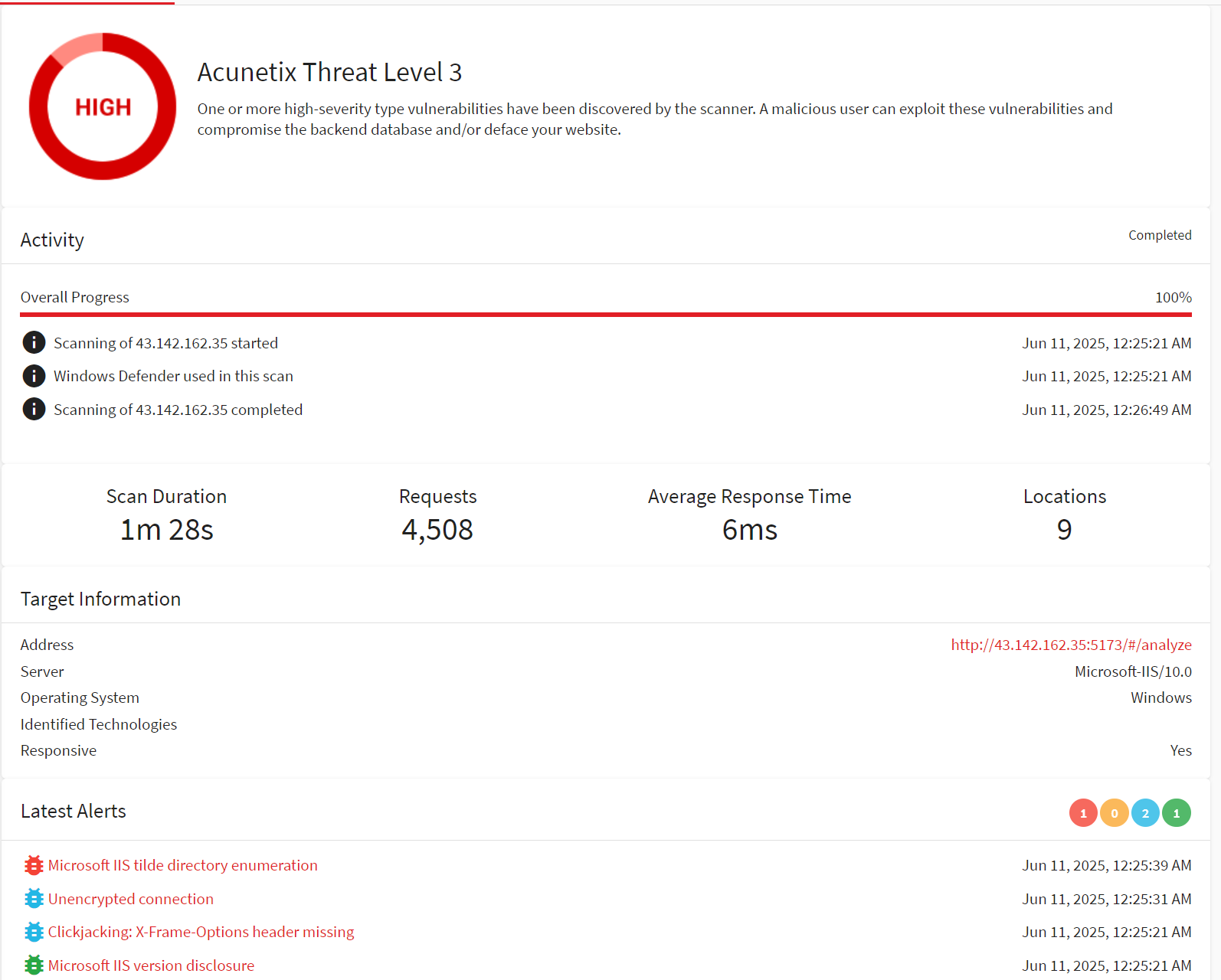


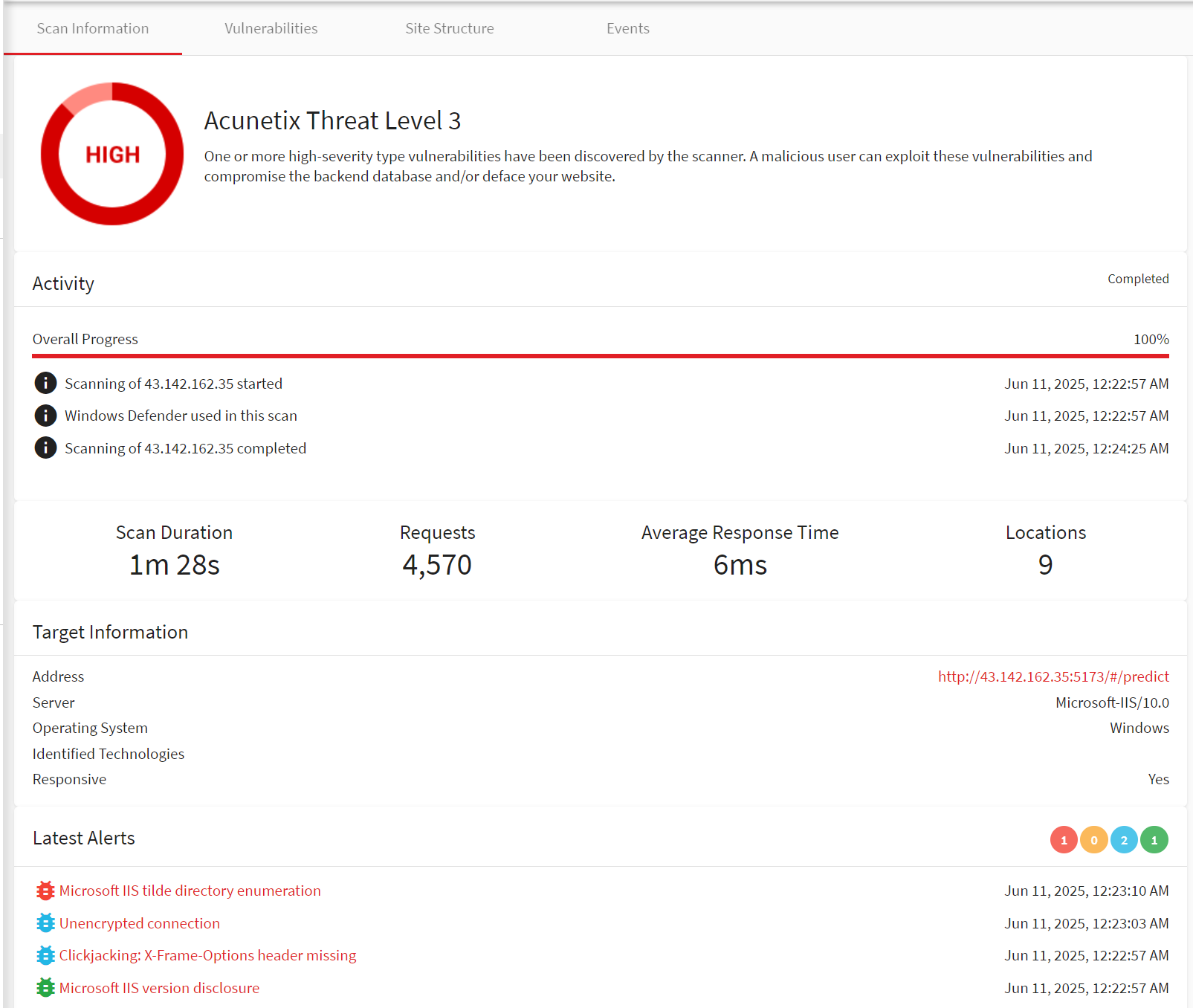
1. TS3-7

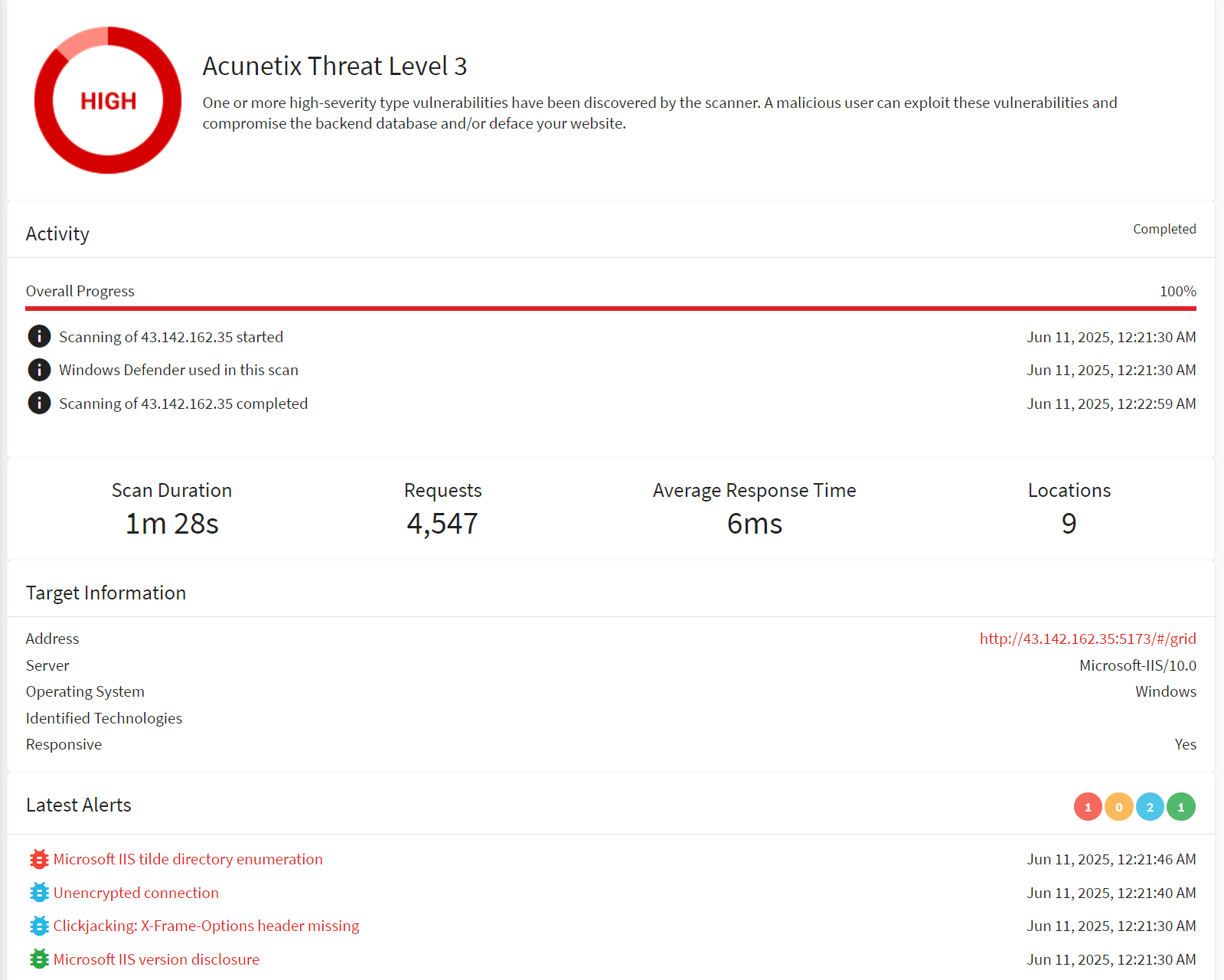


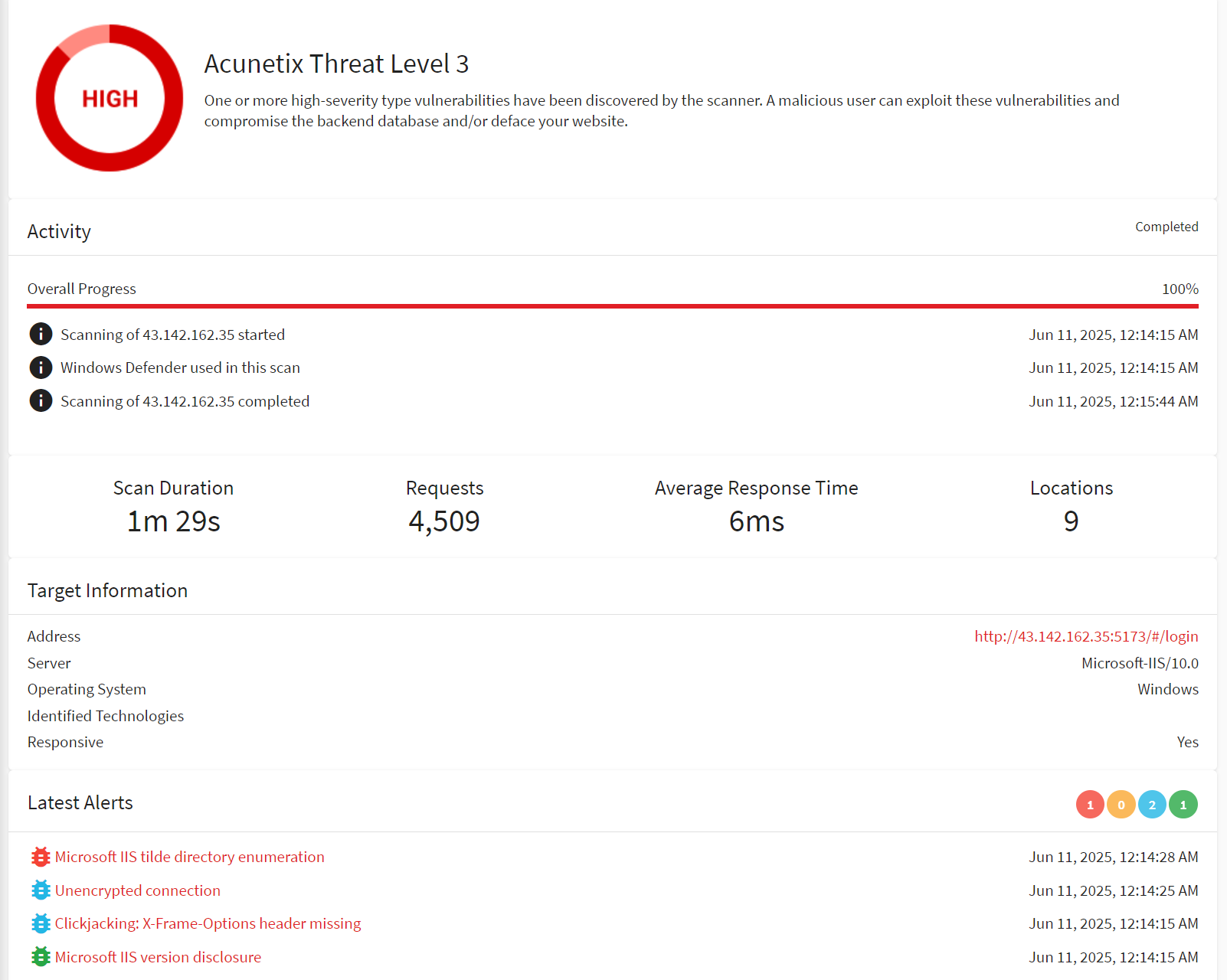
**6.4.4 Acceptance Level Test**











**6.5 Acceptance Decision**

1. Test Scope Review

The acceptance test covered all core business processes, including user authentication, data processing, predictive analysis, and system integration.

1. Test Results Summary

A total of 7 acceptance test cases were executed, 6 passed, and 1 failed due to known defects. All critical paths have been verified.

1. Defect and Risk Assessment

There are 2 low-priority defects, which have been recorded and scheduled for future fixes. No blocking risks identified.

1. Acceptance Conclusion

The system has met the acceptance criteria and is recommended for release.

**7 Appendices**

**7.1 Full Test Case List**

**7.1.1 Component Level Test**

|  |  |
| --- | --- |
| **Main Text Files or Other Files** | **Test Cases** |
| test\_analyze\_service.py | * test\_analyze\_key\_features\_main[y] * test\_analyze\_key\_features\_exceptions[df0-y-ValueError] * test\_analyze\_key\_features\_exceptions[df1-y-ValueError] * test\_analyze\_key\_features\_exceptions[df2-y-ValueError] |
| test\_api\_gateway.py | * test\_api\_gateway\_main |
| test\_auth\_service.py | * test\_auth\_service\_main * test\_auth\_service\_other |
| test\_auth\_utils.py | * test\_auth\_utils\_main * test\_auth\_utils\_other * test\_auth\_utils\_third |
| test\_compare\_service.py | * test\_analyze\_and\_predict\_main[y] * test\_analyze\_and\_predict\_exceptions[df0-y-FileNotFoundError] * test\_analyze\_and\_predict\_exceptions[df1-y-ValueError] * test\_model\_cache * test\_compare\_main |
| test\_data\_service.py | * test\_validate\_data * test\_transform\_data * test\_dms\_str\_to\_float * test\_load\_data\_by\_target |
| test\_db\_utils.py | * test\_db\_utils\_main * test\_db\_utils\_other |
| test\_email\_utils.py | * test\_email\_utils\_main * test\_email\_utils\_other * test\_email\_utils\_third * test\_email\_utils\_fourth |
| test\_excel\_utils.py | * test\_excel\_utils\_main * test\_excel\_utils\_other * test\_excel\_utils\_third |
| test\_preprocess.py | * test\_preprocess\_main * test\_preprocess\_other |
| test\_kan\_utils.py | * test\_kan\_utils\_main * test\_kan\_utils\_other |
| test\_model\_cache.py | * test\_compare\_model\_cache * test\_predict\_model\_cache |
| test\_password\_reset | * test\_password\_reset\_main * test\_password\_reset\_other * test\_password\_reset\_third |
| test\_predict\_service.py | * test\_predict\_service\_main * test\_predict\_service\_other |

**7.1.2 Integration Level Test**

|  |  |
| --- | --- |
| **API** | **Test Cases** |
| * GET 127.0.0.1:8000/api/v1/health | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/auth/login | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/auth/register | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/auth/register/send-code | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/auth/reset-password/send-code | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/auth/reset-password | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/compare/model\_compare | * Correct format * Wrong format |
| * GET 127.0.0.1:8000/api/v1/data/get\_data | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/predict/dependent\_predict | * Correct format * Wrong format |
| * POST 127.0.0.1:8000/api/v1/analyze/dependent\_feature\_analyze | * Correct format * Wrong format |

**7.1.3 System Level Test**

1. Automated testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case Name** | **Script/Tool** | **Main Function/Flow** | **Input/Operation** | **Expected Result** |
| User Register | test\_register.py | Frontend register, email code, uniqueness | email、username、password、verification code | Register success, correct tip |
| User Login | test\_login\_getdata.py | Frontend login, redirect, get data | email/username、password | Login success, redirect |
| Data Analysis | test\_analyze.py | Login, upload Excel, select, analyze, result | email/username、password、file、dependent variable name | Analysis success, result shown |
| Model Compare | test\_compare.py | Login, upload Excel, select, compare, result | email/username、password、file、dependent variable name | Compare success, result shown |
| Model Predict | test\_predict.py | Login, upload Excel, select, predict, result | email/username、password、file、dependent variable name | Predict success, result shown |
| Password Reset | test\_reset\_password.py | Forgot password, email code, update, success tip | email、verification code、new\_password | Password reset success, correct tip |
| Get Data | test\_date.py | Login, select, get data | email/username、password、 | No error, data shown |

1. Concurrency Testing

|  |
| --- |
| **Test Cases** |
| * POST 127.0.0.1:8000/api/v1/auth/login |
| * POST 127.0.0.1:8000/api/v1/auth/register |
| * POST 127.0.0.1:8000/api/v1/auth/register/send-code |
| * POST 127.0.0.1:8000/api/v1/auth/reset-password |
| * POST 127.0.0.1:8000/api/v1/auth/reset-password/send-code |
| * POST 127.0.0.1:8000/api/v1/compare/model\_compare |
| * GET 127.0.0.1:8000/api/v1/data/get\_data |
| * POST 127.0.0.1:8000/api/v1/predict/dependent\_predict |
| * POST 127.0.0.1:8000/api/v1/analyze/dependent\_feature\_analyze |

**7.1.4 Acceptance Level Test**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case Name | Requirement/User Story | Precondition | Steps | Expected Result |
| User Login Acceptance | User can login with username/email and password | Registered user | 1. Open login page 2. Enter username/email and password 3. Click login | Redirect to main page, show user info |
| User Register Acceptance | New user can register and receive code | None | 1. Open register page 2. Enter email, username, password 3. Get and enter verification code 4. Submit registration | Register success, correct tip |
| Data Upload & Analyze Acceptance | User can upload data and analyze | Logged in | 1. Go to analyze page 2. Upload Excel file 3. Select variable 4. Click analyze | Show analysis result table/chart |
| Prediction Acceptance | User can upload data and predict | Logged in | 1. Go to prediction page 2. Upload Excel file 3. Select variable 4. Click predict | Show prediction chart |
| Compare Acceptance | User can compare model results | Logged in | 1. Go to compare page 2. Upload Excel file 3. Select variable 4. Click compare | Show compare chart |
| Password Reset Acceptance | User can reset password with email code | Registered user | 1. Open forgot password page 2. Enter email and new password 3. Get and enter verification code 4. Submit reset | Password reset success, correct tip |
| Usability & UI Acceptance | UI is user-friendly, flow is smooth | None | 1. Browse all pages 2. Operate main functions | No major confusion, clear tips |

**7.2 Links to Related Documents**

1. https://github
2. https://github
3. https://github
4. https://ieeexplore.ieee.org/document/159342
5. https://www.accelq.com/blog/test-coverage-techniques/