



SUNIMOD FIELD NOTES

Unrestricted API Spending: When Your Website Becomes a Blank Check

A public feature can trigger paid API work without exposing a key. Learn how atomic budgets, bounded jobs, and rate limits contain the cost.

BY Christopher Jacobson

PUBLISHED October 15, 2026

TOPIC CISSP, Domain 2

<https://sunimod.com/unrestricted-api-spending-business-cost-controls/>

The business risk in one sentence

A feature that can commit unlimited paid work is an open-ended spending authority disguised as a website button. The fix is to turn a business-approved loss limit into a server-enforced rule: validate the work, reserve its maximum cost atomically, and stop admitting jobs when the available allowance is gone.

[OWASP calls this unrestricted resource consumption](#). Its description includes both exhausted computing resources and increased costs from metered third-party services. No stolen administrator account or exposed vendor key is required for this class of abuse.

In this guide

1. [How ordinary requests become billable work](#)
2. [Why rate limits and alerts are not enough](#)
3. [An enforceable spending policy](#)
4. [A working atomic budget gate](#)
5. [Tests that do not create a provider bill](#)
6. [A supporting NGINX limit](#)
7. [Business repercussions and response](#)
8. [A practical Sunimod project](#)

How ordinary requests become billable work

Consider a fictional customer portal that generates a paid report for each selected record. The browser submits record identifiers. Your application verifies access, calls a reporting provider using a server-held credential, and returns the results. This is useful functionality; the danger is leaving the amount of permitted work undefined.

Here is a normalized example request, not a command to run against a live service. The hostname and credential are fictitious.

Plaintext · an ordinary request that can initiate paid work

```
POST /api/reports HTTP/1.1
Host: portal.example.invalid
Authorization: Bearer fictitious-example-token-not-valid
Content-Type: application/json

{"records":["record_1","record_2","record_3"]}
```

Suppose the backend accepts any length of record list, repeats the provider call for every element, and permits the account to submit more work indefinitely. A person does not need to discover the provider

credential. The application already knows how to authenticate upstream and willingly spends from the business account.

1. **A useful operation is exposed.** A customer-facing route accepts work that creates a metered charge.
2. **The caller satisfies the existing entry conditions.** That might mean using a public form, a free account, or a legitimate customer session.
3. **The backend expands the request.** One submitted batch becomes many provider operations, or one operation starts an expensive processing job.
4. **The caller requests more work than the business intended.** Repeated submissions or oversized batches remain valid because no enforceable allowance says otherwise.
5. **Workers execute using the business account.** Normal application permissions authorize the upstream requests.
6. **The consequences arrive elsewhere.** The website may still look healthy while the provider bill, queue, or shared-resource demand grows.

This is a failure to constrain delegated authority. Authentication answers who is requesting work. Object authorization answers whether they may access a particular record. Neither decision, by itself, answers how much the company is willing to spend serving that account.

It also does not require malicious intent. An integration that repeatedly resubmits a timed-out request can produce the same spending pattern. The control should constrain authorized work regardless of whether the excessive demand came from abuse, a programming mistake, or an unexpected product launch.

One request can create many charges

For this fictional portal, assume each processed record costs exactly \$0.04. A batch contains 25 records, and the service accepts 20 such requests per minute for six hours. The resulting calculation is:

20 requests × 25 records × \$0.04 × 360 minutes = \$7,200.

These are illustrative assumptions, not current vendor prices, observed losses, or a forecast. The calculation excludes hosting, storage, taxes, and other charges.

The incoming rate is only one request every three seconds. Counting requests without counting the work inside them misses the essential cost multiplier. A policy must cover both the size of each operation and the accumulated work admitted across accounts and workers.

There are real versions of this business model. [Twilio describes SMS pumping](#), in which attackers abuse fields that trigger text messages and may receive a share of the resulting telecommunications revenue. A form can therefore be economically attractive to an attacker even when it exposes no customer database.

Why common controls fall short

A rate limit is not a total spending limit

A per-address limiter can slow one source. It does not establish an account-wide financial ceiling across many addresses, application instances, or different expensive routes. A per-account limiter is useful too, but separate accounts can still draw from the same provider balance.

For the portal in this article, use distinct controls for request frequency, maximum records per request, per-account allowance, total feature allowance, and worker concurrency. Each answers a different question. The underlying resource-allocation problem is also described in [MITRE CWE-770](#); unconstrained consumption can deny resources to other applications or users.

An alert is not permission to spend

An email saying the budget has been exceeded is useful evidence, but the spending decision has already happened. [AWS explicitly warns that budget notifications can lag incurred charges](#), allowing additional costs before a notification arrives. AWS also supports [budget actions](#), but a billing-triggered reaction is still different from checking an allowance before admitting each job.

Keep provider alerts and supported provider-side caps as independent safeguards. Verify what each setting actually controls, when it takes effect, and whether accepted or in-progress jobs can still incur charges. Do not label a notification threshold a hard cap in your risk register.

A budget counter can still have a race condition

Checking the balance, calling the provider, and updating the balance afterward is unsafe when workers can overlap. Two requests can each observe the same remaining allowance and both act on it.

Plaintext · an unsafe read-check-spend sequence

```
Available allowance: 40 cents

Worker A reads 40 cents available; its job needs 40 cents.
Worker B reads 40 cents available; its job needs 40 cents.
Worker A decides the job is affordable.
Worker B decides the job is affordable.
Worker A starts 40 cents of paid work.
Worker B starts 40 cents of paid work.

Committed work: 80 cents against a 40-cent allowance.
```

The control must reserve the allowance before paid work begins, with the check and reservation performed as one atomic decision. In the implementation below, [SQLite's immediate transaction](#) obtains a write transaction before checking and updating the shared ledger. Other writers cannot independently spend the same available balance.

Define the spending policy before choosing the limit

Start with an owner and a business decision, not a convenient configuration default. Which feature is allowed to incur charges? What is its legitimate demand? What loss is tolerable? Who can increase its allowance, and what should customers experience when it runs out?

[NIST's risk-management preparation guidance](#) emphasizes defined responsibilities, risk tolerance, assessment, and monitoring. Applied here, that means a product or business owner approves the exposure, engineering implements the boundary, and operations knows how to respond when it closes.

The working example uses a deliberately small, **nonrenewing allowance for one fixed operation**. These are test policy values, not recommended production prices or universal limits.

Illustrative policy implemented by the budget gate

Decision	Enforced value
Maximum work in one request	10 record identifiers
Assumed upper-bound cost per record	4 cents
Account allowance	200 cents across new reservations
Total feature allowance	1,000 cents across all accounts
Repeated request identifier	No second reservation for identical work; reject changed work
Ledger unavailable	Do not authorize new paid work

A reservation is only as accurate as its cost model. Replace the illustrative four-cent bound with a verified maximum for the actual operation. Include every billable sub-operation and any permitted retries. Variable-price services need enforceable bounds on their cost-driving parameters; an average historical charge is not a safe maximum.

For recurring daily or monthly budgets, define rollover and outstanding commitments explicitly. This example does not reset itself at midnight, refund failed requests, or replenish an exhausted account. That conservative choice keeps a time boundary from silently creating a new allowance while earlier work remains unresolved.

Implement a working atomic budget gate

The following complete Python module implements a real local admission ledger. It uses Python's standard library and a SQLite database shared by application processes on one host. It does not call a provider, invent report results, or implement the surrounding HTTP application.

Save it as `budget_gate.py`. Provision the ledger once in a private application-data directory with `initialize()`. For every proposed operation, call `reserve()` after authentication and record-access checks but before creating paid work.

Python · `budget_gate.py` — atomic account and feature allowances

```

"""Atomic, nonrenewing admission budgets for one fixed-price operation."""

from contextlib import closing
import hashlib
import json
from pathlib import Path
import re
import sqlite3
from uuid import UUID

UNIT_BOUND_CENTS = 4
MAX_RECORDS = 10
ACCOUNT_LIMIT_CENTS = 200
FEATURE_LIMIT_CENTS = 1000

class BudgetDenied(RuntimeError):
    """The requested reservation exceeds an applicable budget."""

def initialize(database: Path) -> None:
    """Provision once; existing balances and limits are preserved."""
    with closing(sqlite3.connect(database)) as db:
        with db:
            db.execute("""
                CREATE TABLE IF NOT EXISTS budgets (
                    name TEXT PRIMARY KEY,
                    limit_cents INTEGER NOT NULL CHECK (limit_cents > 0),
                    used_cents INTEGER NOT NULL DEFAULT 0
                    CHECK (used_cents >= 0 AND used_cents <= limit_cents)
                )
            """)
            db.execute("""
                CREATE TABLE IF NOT EXISTS reservations (
                    subject TEXT NOT NULL,
                    request_id TEXT NOT NULL,
                    fingerprint TEXT NOT NULL,
                    cents INTEGER NOT NULL CHECK (cents > 0),
                    created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
                    PRIMARY KEY (subject, request_id)
                )
            """)
            db.execute(
                "INSERT OR IGNORE INTO budgets(name, limit_cents) VALUES (?, ?)",
                ("feature", FEATURE_LIMIT_CENTS),
            )

def reserve(
    database: Path,
    subject: str,
    request_id: str,
    records: list[str],
) -> bool:
    """Return True only for a new reservation; False means duplicate."""
    if not isinstance(subject, str) or not 1 <= len(subject) <= 128:
        raise ValueError("A trusted, nonempty account identifier is required")
    if not isinstance(request_id, str):
        raise ValueError("request_id must be a UUID string")
    request_id = str(UUID(request_id))
    if not isinstance(records, list) or not 1 <= len(records) <= MAX_RECORDS:
        raise ValueError("Provide between 1 and 10 record identifiers")
    if any(
        not isinstance(record, str)
        or re.fullmatch(r"[A-Za-z0-9-]{1,64}", record) is None
        for record in records
    ):
        raise ValueError("Invalid record identifier")

    payload = json.dumps(
        {"operation": "fixed-report-v1", "records": records},
        sort_keys=True,
        separators=(",", ":"),

```

```

        allow_nan=False,
    )
    fingerprint = hashlib.sha256(payload.encode("utf-8")).hexdigest()
    cents = len(records) * UNIT_BOUND_CENTS
    account = "account:" + subject

    with closing(sqlite3.connect(
        database, timeout=5, isolation_level=None
    )) as db:
        db.execute("BEGIN IMMEDIATE")
        try:
            previous = db.execute(
                """SELECT fingerprint FROM reservations
                   WHERE subject = ? AND request_id = ?""",
                (subject, request_id),
            ).fetchone()
            if previous is not None:
                if previous[0] != fingerprint:
                    raise ValueError("Request identifier reused with different work")
                db.execute("COMMIT")
                return False

            db.execute(
                "INSERT OR IGNORE INTO budgets(name, limit_cents) VALUES (?, ?)",
                (account, ACCOUNT_LIMIT_CENTS),
            )
            for bucket in ("feature", account):
                result = db.execute(
                    """UPDATE budgets SET used_cents = used_cents + ?
                       WHERE name = ? AND used_cents <= limit_cents - ?""",
                    (cents, bucket, cents),
                )
                if result.rowcount != 1:
                    raise BudgetDenied("Budget exhausted: " + bucket)

            db.execute(
                """INSERT INTO reservations
                   (subject, request_id, fingerprint, cents)
                   VALUES (?, ?, ?, ?)""",
                (subject, request_id, fingerprint, cents),
            )
            db.execute("COMMIT")
            return True
        except BaseException:
            if db.in_transaction:
                db.execute("ROLLBACK")
            raise

```

What this code enforces

The function validates the batch, calculates a server-controlled reservation, and binds the request identifier to the account and complete record list for this fixed operation. Both allowance updates and the reservation record commit together. If either allowance is insufficient, the transaction rolls back both updates.

A duplicate submission returns `False`, not a second authorization. Changing the record list while reusing the same account and request identifier raises an error. The fingerprint is a consistency check; it is not a credential, signature, or replacement for authorization.

The transaction is finished before control returns to the application. Never keep this write transaction open while waiting for a network provider. The explicit transaction behavior follows the [Python SQLite interface](#) and SQLite's transaction model.

The integration contract matters as much as the counter

Identity must come from authenticated server context. The `subject` value must not be copied from an arbitrary request-body account field. Validate access to every record first. This gate limits consumption; it does not establish who owns a record or implement authentication.

Only a new reservation may initiate new paid work. A `True` result permits the caller to initiate that bounded operation once. A `False` result means the application must retrieve the existing operation's status rather than submit again. It does not mean the report completed successfully.

The actual job must match the reserved work. This example has one fixed report operation and a bounded list of records. A production job must not add destinations, larger batches, alternate models, or additional provider calls after reservation. Include every behavior-changing field in the validated job representation and cost calculation.

Uncertain outcomes must retain their reservation. A timeout does not prove the provider did nothing. Refunding the allowance immediately and retrying can fund duplicate work. Durable job state, provider-supported idempotency, and reconciliation are separate requirements. [Amazon's guidance on idempotent APIs](#) explains why request identifiers and retry semantics must be designed around side effects.

Failures must close the gate. Treat a locked, missing, corrupt, or unreachable ledger as unavailable admission control. Do not catch its database exception and continue with the provider request. A budget rejection can produce a documented quota response; an unavailable control should produce a service-unavailable response without exposing database details.

Deployment boundary: this is a single-host admission component, not a distributed billing platform or an exactly-once job executor. Every paid entry path must use the same authoritative ledger. Separate database copies on different servers create separate allowances. Protect the database from deletion or rollback, and reconcile provider commitments before recovering an old copy. For multiple application hosts, implement equivalent atomic reservations in a shared transactional service.

This ledger bounds reservations for the modeled operation, not the entire cloud bill. Gateway traffic, database work, storage, and provider activity outside this path need their own controls. A depleted allowance can also deny legitimate work, so decide which essential workflows require separately protected capacity.

For this design, keep a durable operation record containing the validated job, reservation, provider request identifier, and outcome. Coordinate that record with queue admission so a crash cannot turn a duplicate delivery into an unreserved provider call. Retain unresolved reservations until the outcome is known; do not delete the ledger to restore service.

Test the limit without creating a provider bill

These tests exercise the actual ledger, including simultaneous connections competing for the same allowance. They create temporary local databases and contact no external service. Save the file as `test_budget_gate.py` beside the module.

Python · test_budget_gate.py — bounded local verification

```

from concurrent.futures import ThreadPoolExecutor
from contextlib import closing
from pathlib import Path
import sqlite3
import tempfile
import unittest
from uuid import uuid4

from budget_gate import BudgetDenied, initialize, reserve

class BudgetTests(unittest.TestCase):
    def setUp(self):
        directory = tempfile.TemporaryDirectory()
        self.addCleanup(directory.cleanup)
        self.database = Path(directory.name) / "budget.sqlite3"
        initialize(self.database)

    def used(self):
        with closing(sqlite3.connect(self.database)) as db:
            return db.execute(
                "SELECT used_cents FROM budgets WHERE name = 'feature'"
            ).fetchone()[0]

    def admit(self, subject, request_id=None, records=None):
        return reserve(
            self.database,
            subject,
            str(uuid4()) if request_id is None else request_id,
            ["record_1"] if records is None else records,
        )

    def test_duplicates_and_changed_payload(self):
        key = str(uuid4())
        self.assertTrue(self.admit("account_a", key))
        self.assertFalse(self.admit("account_a", key))
        with self.assertRaises(ValueError):
            self.admit("account_a", key, ["record_2"])
        self.assertEqual(self.used(), 4)

    def test_validation(self):
        for records in ([], ["record_1"] * 11, ["https://example.invalid/"]):
            with self.assertRaises(ValueError):
                self.admit("account_a", records=records)
        self.assertEqual(self.used(), 0)

    def test_account_limit_rolls_back_feature_debit(self):
        for _ in range(5):
            self.assertTrue(self.admit("account_a", records=["record_1"] * 10))
        with self.assertRaises(BudgetDenied):
            self.admit("account_a")
        self.assertEqual(self.used(), 200)
        initialize(self.database)
        self.assertEqual(self.used(), 200)

    def test_concurrent_feature_limit(self):
        def attempt(index):
            try:
                return self.admit(
                    f"account_{index}", records=["record_1"] * 10
                )
            except BudgetDenied:
                return False

        with ThreadPoolExecutor(max_workers=8) as pool:
            results = list(pool.map(attempt, range(40)))
        self.assertEqual(sum(results), 25)
        self.assertEqual(self.used(), 1000)

```

```
def test_concurrent_duplicate(self):
    key = str(uuid4())
    with ThreadPoolExecutor(max_workers=8) as pool:
        results = list(pool.map(
            lambda _: self.admit("account_a", key), range(20)
        ))
    self.assertEqual(sum(results), 1)
    self.assertEqual(self.used(), 4)

def test_missing_ledger_raises(self):
    with self.assertRaises(sqlite3.OperationalError):
        reserve(
            self.database.parent / "missing.sqlite3",
            "account_a",
            str(uuid4()),
            ["record_1"],
        )

if __name__ == "__main__":
    unittest.main()
```

Run the suite in an Ubuntu terminal from the directory containing both files:

Bash · run the six local tests

```
python3 -m unittest -v test_budget_gate.py
```

The supplied files passed all six tests during preparation, including the 40-request concurrency check. Twenty-five reservations of 40 cents were admitted; the remaining requests could not overdraw the 1,000-cent feature allowance. Concurrent duplicates produced one reservation, and an account-limit rejection did not leave a partial feature debit.

The useful evidence is the invariant: admitted commitments remain within both applicable allowances even when requests overlap. These tests verify the admission component, not your authentication, queue delivery, provider pricing, or production failover configuration.

For an integration assessment, add a negative test proving that the provider adapter is never invoked after denial or a database error. Verify restart behavior, duplicate queue delivery, ambiguous provider timeouts, and price-policy changes in an authorized non-billable environment before enabling live traffic.

Add an edge limit without mistaking it for a budget

Request throttling still has a job: reduce repeated traffic before it consumes application and database capacity. [NGINX supports per-key request-rate limits](#), including rates measured per minute and explicit rejection status codes.

This configuration belongs in an NGINX `http` context. It is a loopback-only example for an existing Ubuntu application listening on port 8000, not a public HTTPS deployment. The small limits are illustrative.

NGINX · loopback gateway with a request-rate boundary

```
limit_req_zone $binary_remote_addr zone=report_per_ip:10m rate=6r/m;
```

```
server {
    listen 127.0.0.1:8080;
    server_name portal.example.invalid;
    client_max_body_size 16k;

    location = /api/reports {
        limit_req zone=report_per_ip burst=2 nodelay;
        limit_req_status 429;

        proxy_connect_timeout 3s;
        proxy_send_timeout 10s;
        proxy_read_timeout 20s;
        proxy_next_upstream off;

        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_pass http://127.0.0.1:8000;
    }

    location / {
        return 404;
    }
}
```

The limiter permits an average six requests per minute per client address, with a small burst allowance; it is not a strict six-request calendar-minute counter. Local requests share the loopback address. In a public deployment behind a proxy or CDN, configure trusted client-address handling rather than trusting arbitrary forwarded headers.

The configuration also bounds request-body size and disables NGINX's upstream failover retries for this route. Its [proxy timeouts](#) do not prove that upstream work stopped or that a provider canceled billing. The body-size setting is documented in the [NGINX core module](#). Application and queue retry policies still need their own controls.

At the application boundary, validate content type, payload shape, allowed methods, record ownership, and maximum batch size. [OWASP's REST security guidance](#) covers these input and endpoint controls. Do not rely on a disabled browser button or client-side validation to enforce spending authority.

Potential repercussions for your business and infrastructure

The impact depends on the service's prices, available allowance, processing limits, and operational dependencies. For the fictional portal, assess at least these consequences rather than treating the issue as merely a noisy endpoint.

- **Unplanned expenditure.** Paid operations consume funds without corresponding customer value. Include investigation and reconciliation work in the incident estimate, not just provider charges.
- **Unavailable customer features.** Exhausted quotas or a necessary emergency cutoff can block legitimate report generation, onboarding, or notifications.
- **Resource contention.** Heavy jobs can occupy workers and shared infrastructure needed by unrelated features. Resource starvation is a recognized consequence of [unbounded allocation](#).
- **Delayed recovery.** A backlog of already accepted jobs can remain after incoming requests stop. The team must distinguish work that is queued, started, canceled, or already charged.

- **Customer frustration and support load.** Duplicate results, missing messages, slow responses, or unexplained quota denials can create avoidable complaints and manual recovery tasks.
- **Distorted business decisions.** Abuse mixed into usage totals can make demand appear stronger than it is. Separate admitted work, completed work, abuse signals, and reconciled charges before using those numbers for product planning.

Cost abuse does not, by itself, prove that confidential data was stolen. Investigate access and data exposure separately rather than describing every expensive incident as a data breach.

Respond without turning the cleanup into more spending

First, pause new paid admissions for the affected operation. Preserve essential customer paths that use separate allowances. Pausing the public route alone is insufficient when background workers are still draining an existing queue.

Next, preserve the admission ledger, operation identifiers, queue state, timestamps, and provider receipts. Record which jobs can be canceled and which have uncertain outcomes. Avoid automatic refunds or replays while reconciling those states. Do not log credentials or unnecessarily copy sensitive payloads into the investigation record.

Finally, compare reservations with provider-reported usage, close any path that bypassed admission, and verify the upper-bound cost model. Rotating a credential is appropriate when exposure is suspected, but a replacement key will not fix an application that still accepts unlimited paid work.

Turn the weakness into a scoped Sunimod project

The solvable business idea is a **metered-feature control layer**: connect each charge-producing workflow to an owner, an enforceable allowance, and evidence showing why work was admitted. That can be a focused improvement to an existing website or a requirement built into a new application.

Hire Sunimod to assess and scope that improvement around your actual workflow. A useful project definition is not simply “add rate limiting.” It specifies the paid operations, the allowed exposure, the shared admission mechanism, the customer experience at the limit, and how the business verifies the result.

- **Inventory the spending paths:** user-facing routes, scheduled tasks, worker retries, administrative tools, and third-party integrations.
- **Approve the policy:** cost bounds, account and feature allowances, exception authority, and a named operational owner.
- **Implement the controls:** validated jobs, atomic reservations, bounded retries, durable operation tracking, and layered request limits.
- **Prove the boundary:** concurrency tests, duplicate handling, failure-closed behavior, and reconciliation against provider activity.
- **Hand over an operating procedure:** alerts, pause controls, recovery steps, and a review process for changing prices or feature behavior.

Measure success with concrete evidence: coverage of paid entry paths, denied work that generated no provider call, unresolved commitments, and the difference between reserved and reconciled charges. A provider change or a new job type should trigger a policy review, not silently inherit an unrelated allowance.

The goal is not to make useful automation inconvenient. It is to make the cost of that automation a deliberate business decision instead of an unlimited permission inherited by every accepted request.

Sources and further reading

- [OWASP: API4:2023 Unrestricted Resource Consumption](#) — metered-service abuse and resource limits.
- [MITRE: CWE-770](#) and [OWASP: REST Security Cheat Sheet](#) — allocation boundaries and endpoint validation.
- [NIST: Risk Management Framework Prepare Step](#) — ownership, risk tolerance, and monitoring.
- [AWS: Managing your costs with AWS Budgets](#) and [Configuring a budget action](#) — notification timing and reactive controls.
- [Amazon Builders' Library: Making retries safe with idempotent APIs](#) — request identity and side effects.
- [SQLite: Transactions](#) and [Python: sqlite3](#) — transaction semantics used by the implementation.
- [NGINX: Request limiting, proxying, and request-body limits](#) — supporting gateway controls.
- [Twilio: What is SMS pumping fraud?](#) — an example of economically motivated abuse of metered features.

The report portal, credentials, prices, and allowances are fictional teaching examples. The admission module and tests are original examples; the linked documentation supports the technical concepts, not an endorsement of this implementation.

Put a boundary around your paid automation

Does your website trigger paid reports, messages, processing, or other API work? Hire Sunimod to help define and implement a spending-control project for that workflow. Describe the feature, its provider integrations, and the exposure you need to contain; do not include passwords, access tokens, or API keys.

[Request a Sunimod project quote](#)