

You are now the Principal Engineer performing a production-readiness audit of this reusable product module.

DO NOT rewrite the entire project blindly.

Inspect every file.

Audit:

1. Architecture
2. Type safety
3. Authentication
4. Authorization
5. Tenant isolation
6. Database design
7. Transactions
8. Race conditions
9. Concurrency
10. API security
11. Input validation
12. Output validation
13. CSRF
14. XSS
15. SSRF
16. SQL injection
17. IDOR
18. Privilege escalation
19. Session security
20. Token security
21. Rate limiting
22. File security
23. Webhook security
24. Payment idempotency
25. AI security
26. Prompt injection
27. Secret leakage
28. Logging
29. Privacy
30. Error handling
31. Performance
32. Caching
33. Database indexes
34. N+1 queries
35. Background jobs
36. Retry behavior
37. Failure recovery
38. Accessibility

- 39. SEO
- 40. Testing
- 41. Deployment
- 42. Environment variables
- 43. Backup/recovery
- 44. Dependency vulnerabilities
- 45. Production configuration

For every problem found:

SEVERITY:
CRITICAL / HIGH / MEDIUM / LOW

FILE:
exact file

PROBLEM:
what is wrong

RISK:
what could happen

FIX:
specific fix

TEST:
how to verify the fix

Then fix all CRITICAL and HIGH issues.

Fix MEDIUM issues where practical.

Do not introduce unnecessary complexity.

Do not replace working architecture merely for stylistic reasons.

After fixing:

- run tests
- run type checking
- run linting
- run build
- run security checks

Then perform a second audit of the changed code.

Finally produce:

1. Architecture summary
2. Security summary
3. Performance summary
4. Remaining risks
5. Production deployment checklist
6. Environment variable checklist
7. Backup checklist
8. Monitoring checklist
9. Future improvements

Do not claim "100% secure" or "zero vulnerabilities".

The goal is a defensible, production-grade implementation with clearly identified remaining risks.

You are a Principal Software Architect, Staff+ Full-Stack Engineer, Application Security Engineer, DevOps Engineer, Database Architect, SRE, QA Engineer and UI/UX Engineer.

I am building a reusable production-grade module for my personal PRODUCT FACTORY.

IMPORTANT ARCHITECTURAL RULE:

This module must be completely independent.

It must NOT assume that 50–100 products live inside one centralized application.

Every product I build will have:

- its own repository
- its own database
- its own deployment
- its own environment variables
- its own users
- its own authentication
- its own billing
- its own admin dashboard
- its own analytics
- its own product-specific data

This module is being designed as a reusable component that I can copy, install or adapt into each independent product.

Therefore:

DO NOT create a centralized multi-product SaaS platform.

DO NOT create cross-product dependencies.

DO NOT require another product to exist.

DO NOT require a central database.

DO NOT hard-code provider-specific business logic.

Make the module self-contained, modular and portable.

TECHNICAL QUALITY REQUIREMENTS:

- TypeScript
- strict type safety
- production-ready architecture
- secure by default
- scalable
- testable
- maintainable
- accessible
- performant
- observable
- documented
- graceful failure handling
- idempotent where required
- proper database constraints
- proper indexes
- transaction safety
- input validation
- authorization checks
- rate limiting where appropriate
- no secrets in source code
- no sensitive information in logs
- secure error handling
- current stable dependencies
- avoid abandoned packages
- use official SDKs where possible

Before writing code:

1. Inspect the existing repository.
2. Understand the existing framework.
3. Detect package manager.
4. Detect database.

5. Detect existing authentication.
6. Detect existing UI system.
7. Detect existing environment configuration.
8. Reuse existing infrastructure where appropriate.
9. Do not unnecessarily rewrite existing code.
10. Do not introduce dependencies without justification.

First produce an implementation architecture.

Then implement it in small logical stages.

After implementation:

- run type checking
- run linting
- run unit tests
- run integration tests where appropriate
- run security checks
- fix all errors
- document installation
- document configuration
- document environment variables
- document API usage
- document edge cases
- document deployment

Do not claim something is production-ready unless the implementation and tests actually support that claim.

01 — UNIFIED AUTHENTICATION & USER MANAGEMENT

Build a completely reusable authentication module.

It must support:

- email/password
- Google OAuth
- GitHub OAuth
- Apple OAuth
- magic links
- email verification
- password reset
- account recovery
- MFA/2FA
- session management
- logout
- logout all devices

- session revocation
- device/session listing
- secure token handling
- JWT/session abstraction where appropriate

Create reusable UI components:

```
<LoginForm />
<SignupForm />
<MagicLinkForm />
<OAuthButtons />
<VerifyEmail />
<ResetPassword />
<MFASetup />
<MFAChallenge />
<UserProfile />
<SessionManager />
```

Security requirements:

- secure password hashing
- secure cookies
- HttpOnly
- Secure
- appropriate SameSite
- CSRF protection where applicable
- brute-force protection
- login rate limiting
- account enumeration protection
- secure token expiration
- refresh-token rotation where applicable
- session invalidation
- secure OAuth state handling
- PKCE where appropriate
- prevent session fixation
- prevent privilege escalation

Create clean interfaces so the underlying provider can later be changed.

Possible providers:

- Supabase Auth
- Clerk
- Auth.js
- custom implementation

Do not tightly couple application business logic to the provider.

Create:

auth/

- |— server/
- |— client/
- |— components/
- |— middleware/
- |— providers/
- |— schemas/
- |— tests/
- |— docs/

Include complete tests for:

- signup
- login
- logout
- OAuth
- magic link
- password reset
- MFA
- session expiration
- revocation
- unauthorized access
- rate limiting
- CSRF
- account enumeration.

=====

SERVER-SIDE VALIDATION — NON-NEGOTIABLE

=====

NEVER trust client-side validation.

Every authentication-related endpoint must independently validate all incoming data on the server.

Use a strict schema-validation layer such as Zod or an equivalent well-maintained validation library.

Validate:

- request body
- query parameters
- route parameters
- headers where applicable
- cookies
- OAuth callback parameters

- tokens
- redirect URLs
- MFA codes
- password-reset tokens
- invitation tokens

Create explicit schemas:

SignupSchema
LoginSchema
MagicLinkSchema
PasswordResetRequestSchema
PasswordResetSchema
MFASetupSchema
MFACHallengeSchema
OAuthCallbackSchema
SessionSchema

Reject:

- malformed input
- unexpected fields where appropriate
- oversized input
- invalid types
- invalid formats
- expired tokens
- revoked tokens
- reused single-use tokens
- invalid OAuth state
- invalid redirect URLs

Never rely on:

HTML validation
React validation
client-side JavaScript validation
TypeScript types alone
hidden form fields
frontend authorization checks

TypeScript types disappear at runtime and therefore DO NOT constitute server-side validation.

=====

SERVER-SIDE AUTHORIZATION

=====

Authentication and authorization are separate.

Authentication answers:

"Who is this user?"

Authorization answers:

"Is this user allowed to perform this action?"

Every protected server endpoint must perform both where applicable.

Example:

```
authenticateRequest()
  ↓
getAuthenticatedUser()
  ↓
requirePermission()
  ↓
validateRequest()
  ↓
executeBusinessLogic()
```

Never accept:

userId
organizationId
role
permissions
subscription status

from the client as trusted values.

Derive sensitive identity and authorization information from the authenticated server-side session/database.

=====

SERVER-SIDE RESPONSE VALIDATION

=====

Where appropriate, validate structured responses before returning them to clients.

Never accidentally return:

password hashes
refresh tokens
private OAuth credentials

internal secrets
API keys
security-sensitive internal fields

Create explicit public response schemas.

Example:

PublicUserSchema

rather than returning the complete database User object.

=====

AUTHENTICATION SECURITY TESTS

=====

Create automated tests proving that:

1. Invalid email is rejected server-side.
2. Invalid password is rejected server-side.
3. Missing fields are rejected.
4. Unexpected/malicious input is rejected.
5. Oversized requests are rejected.
6. Expired sessions are rejected.
7. Revoked sessions are rejected.
8. Invalid JWTs are rejected.
9. Modified JWT payloads are rejected.
10. Invalid OAuth state is rejected.
11. Invalid OAuth redirect is rejected.
12. Expired password-reset tokens are rejected.
13. Reused password-reset tokens are rejected.
14. Invalid MFA codes are rejected.
15. Excessive login attempts are rate limited.
16. Unauthorized users cannot access protected routes.
17. Users cannot modify another user's account.
18. Users cannot modify their own role/permissions.
19. Admin endpoints reject ordinary users.
20. Sensitive authentication data never appears in API responses.
21. Authentication failures do not leak whether an account exists where account enumeration protection is required.
22. Session invalidation actually prevents further access.
23. Logout invalidates the appropriate session.
24. Logout-all-devices invalidates all applicable sessions.
25. Password reset invalidates appropriate existing sessions.

Additional Requirements:

Identity lifecycle

Account state machine:

PENDING_VERIFICATION

ACTIVE

SUSPENDED

LOCKED

DEACTIVATED

DELETED

Prevent authentication when account state doesn't permit it.

Distinguish:

disabled account

deleted account

unverified account

temporarily locked account.

Account deletion should have a defined grace period if desired.

Re-registration behavior for previously deleted emails must be explicitly defined.

Session lifecycle

Rotate session identifier after authentication.

Rotate/revoke sessions after:

password change

email change

MFA enrollment

MFA removal

suspicious security event

privilege escalation.

Session inactivity timeout.

Absolute session lifetime.

Sliding session expiration where appropriate.

Detect impossible/stale sessions.

Session version/revision mechanism for global invalidation.

Cookie security

Explicitly enforce:

HttpOnly

Secure

appropriate SameSite

narrowly scoped Domain

narrowly scoped Path

expiration/max-age.

Never put authentication credentials in localStorage by default.

Prevent session tokens from appearing in URLs.

Password security

Password strength policy based on configurable rules.

Password confirmation must be validated server-side.

Password change requires current-password verification where appropriate.

Prevent reuse of recent passwords if desired.

Password breach detection integration should be provider-abstracted.

Never log passwords.

Never expose password hashes through serialization.

Password hashing configuration must be upgradeable.

Existing hashes should be transparently rehashed when security parameters improve.

Email change security

Email change should be treated as a sensitive operation:

- authenticate existing session

- reauthenticate if required

- verify new email

- optionally notify old email

- invalidate/revoke pending old verification flows

- update identity atomically.

- Prevent email-change token reuse.

Account recovery

Recovery needs its own threat model:

- recovery-code generation

- recovery-code hashing/storage

- one-time recovery-code usage

- recovery-code regeneration invalidates old codes

- recovery-code rate limits

- recovery attempts audit trail

- recovery should not accidentally bypass MFA without an explicit recovery policy.

MFA

Add:

- replay prevention for MFA codes

- clock-drift handling

- recovery codes

- recovery-code hashing

- enrollment confirmation before activation

- MFA removal requiring stronger authentication

- notification when MFA is changed

- session invalidation after MFA changes

- protection against repeatedly testing codes.

OAuth

Explicitly cover:

- state

- nonce

- PKCE

- exact redirect URI validation

- provider account linking attacks

- email/account collision handling

- untrusted profile fields

- provider identity vs email identity distinction

- unlinking the final authentication provider must not lock the user out.

- OAuth account linking should require authenticated intent.

Enumeration protection

Ensure consistent behavior for:

- signup

- login

- password reset

- magic link

- email verification

account recovery.

Avoid revealing whether an account exists where that matters.

Authentication abuse detection

Add:

IP reputation signals

account-based throttling

device/fingerprint signals where legally appropriate

suspicious login detection

new-device notification

unusual-location notification

security-event escalation.

Do not automatically lock accounts solely because of one noisy signal.

Authorization boundary

Create a formal server pipeline:

Parse request

→ Validate schema

→ Authenticate

→ Load authoritative identity

→ Resolve tenant/context

→ Authorize

→ Execute

→ Validate output

→ Audit

→ Respond

This should become a platform-wide standard, not just an Auth feature.

02 — PAYMENT GATEWAY & SUBSCRIPTION ENGINE

Build a provider-independent billing system.

Support:

- free plans
- paid plans
- subscriptions
- one-time payments
- trials
- coupons
- upgrades
- downgrades
- cancellation
- scheduled cancellation
- pause/resume
- failed payments
- refunds
- invoices
- taxes
- billing portal

- payment history

Create:

BillingProvider

and adapters for:

Stripe

Paddle

Lemon Squeezy

Application code should use:

getSubscription()

hasActivePlan()

hasFeature()

getPlanLimit()

cancelSubscription()

createCheckout()

Never expose provider-specific logic to application business logic.

Webhook system must provide:

- signature verification

- idempotency

- retries

- replay protection

- event persistence

- event processing state

- failure handling

- audit logging

Prevent:

double subscription

double crediting

double refunds

duplicate webhook processing

tampered webhook requests

Create a minimal billing UI:

Pricing

Checkout

Subscription

Billing Portal

Invoices
Payment History

Financial state machine

Define explicit states for:

payment

invoice

subscription

refund

dispute

chargeback

payment method.

Never infer financial state merely from the frontend.

Money representation

Never use floating-point arithmetic for money.

Store integer minor units where appropriate.

Store currency explicitly.

Handle zero-decimal currencies.

Handle currency conversion separately from transaction amount.

Preserve original transaction currency.

Price immutability

Historical invoices must retain the price/tax/discount values that existed when purchased.

Never dynamically recalculate an old invoice from the current plan configuration.

Subscription race conditions

Handle simultaneous:

upgrade + cancellation

cancellation + renewal

payment + downgrade

webhook + user action

duplicate webhook deliveries

delayed webhooks arriving out of order.

Webhook ordering

Events may arrive:

Event B

Event A

Event B again

Do not assume chronological delivery.

Maintain:

event IDs

event timestamps

provider object versions where available

processed-event records

state reconciliation.

Entitlements

Separate:

Payment status

↓

Subscription state



Product entitlement

A product should not scatter:

if paid === true

throughout the application.

Create an entitlement resolver.

Grace periods

Define behavior for:

failed payment

expired card

temporary provider outage

disputed payment

canceled subscription

expired trial.

Tax

Add architecture for:

tax-inclusive/exclusive pricing

billing address

tax IDs

tax calculation provider

tax exemption

invoice tax snapshots.

Refund consistency

Refunding money must correctly update:

payment state

entitlement state if applicable

invoice state

usage/credits if applicable

affiliate commissions

audit records.

Financial reconciliation

Add a reconciliation job comparing:

Internal database

vs

Payment provider

to detect drift.

Double fulfillment prevention

Never grant:

credits

subscription access

downloads

product entitlement

twice because two payment events race.

Billing portal security

Portal/session URLs must be:

generated server-side

short-lived where appropriate

associated with the correct customer

never accepted directly from client input.

Include comprehensive tests.

03 — MULTI-TENANCY & WORKSPACE MANAGEMENT

Build reusable workspace/team infrastructure.

Support:

- organizations
- workspaces
- team members
- invitations
- roles
- permissions
- ownership transfer
- member removal
- role changes

Default roles:

OWNER
ADMIN
MEMBER
VIEWER

Create permission APIs:

can()
canAccess()
requirePermission()

Every workspace-owned resource must support tenant isolation.

Never trust workspace_id supplied directly by the client.

Authorization must happen server-side.

Prevent:

IDOR
cross-tenant access
privilege escalation
unauthorized role changes
unauthorized workspace deletion

Create reusable UI:

WorkspaceSwitcher
InviteMember
MemberList
RoleSelector
WorkspaceSettings

Use database constraints and indexes.

Tenant resolution

Define exactly how tenant context is resolved:

subdomain

custom domain

session

URL

explicit workspace selector

API key

and establish precedence.

Never allow two competing tenant sources to silently disagree.

Tenant context poisoning

Prevent:

authenticated user → tenant A

request → tenant B

from accidentally accessing tenant B.

Membership lifecycle

Add:

invitation expiration

invitation revocation

invitation reuse policy

invitation ownership

pending membership state

member removal

member suspension

ownership transfer

last-owner protection.

Ownership transfer

Require strong verification for:

changing owner

removing owner

transferring billing ownership.

Tenant deletion

Define:

confirmation

grace period

asynchronous deletion

dependent data deletion

file deletion

webhook/API-key revocation

job cancellation

domain detachment

backup retention.

Cross-tenant cache isolation

Cache keys must include tenant identity.

Example:

tenant:{tenantId}:resource:{resourceId}

Never:

resource:{resourceId}

when IDs aren't globally isolated.

Background jobs

Every tenant-aware job must carry immutable tenant context.

Never rely on:

currentUser

inside asynchronous workers.

Database enforcement

Where possible, add database-level tenant constraints in addition to application checks.

Include exhaustive authorization tests.

04 — TRANSACTIONAL EMAIL & NOTIFICATION CENTER

Build a unified notification system.

Support:

EMAIL

IN_APP

PUSH

WEBHOOK

Create:

sendNotification()

with:

userId

template

data

channel

priority

Support:

welcome emails

verification
password reset
magic links
invoices
payment failures
subscription events
system alerts

Features:

- templates
- localization
- retry queue
- idempotency
- unsubscribe preferences
- notification preferences
- digest mode
- delivery tracking
- failure tracking

Create provider abstraction for:

Resend
SendGrid
AWS SES

Create reusable components:

NotificationCenter
NotificationBell
NotificationList
NotificationPreferences

Ensure provider failure does not crash core application workflows.

Notification state machine

Track:

QUEUED
PROCESSING
SENT
DELIVERED
BOUNCED
FAILED
COMPLAINED
CANCELED

rather than only sent=true.

Provider failover

Define what happens when the email provider is unavailable.

Possible:

primary provider

→ retry

→ fallback provider

→ dead-letter

Email deduplication

Prevent accidental duplicate emails caused by:

retries

HTTP timeouts

worker crashes

duplicate events.

Email rendering safety

HTML escaping

template injection prevention

URL validation

user-controlled content sanitization

safe link generation.

Email security

Sensitive links must have:

expiration

one-time use

purpose binding

user/account binding

no sensitive information in URLs beyond what is necessary.

Bounce management

Handle:

hard bounce

soft bounce

spam complaint

suppression list.

Avoid repeatedly sending to permanently invalid addresses.

Unsubscribe semantics

Separate:

transactional messages

marketing messages

security messages

legally required communications.

Do not accidentally unsubscribe users from security-critical messages.

Timezone-aware delivery

Scheduled notifications should respect user timezone and DST.

Digest correctness

Avoid:

duplicate items

missing items

already-read content

race conditions between digest generation and event creation.

05 — SETTINGS, PREFERENCES & CUSTOMIZATION

Build a complete reusable settings system.

Sections:

Profile
Account
Security
Notifications
Appearance
Language
Privacy
Billing
Connected Accounts

Support:

- name
- avatar
- timezone
- locale
- theme
- notification preferences
- privacy preferences

Create typed settings schemas.

Do not create uncontrolled arbitrary JSON where structured schemas are appropriate.

Create:

SettingsLayout
ProfileSettings
SecuritySettings
NotificationSettings
AppearanceSettings
PrivacySettings

Every setting must have server-side validation and authorization.

Sensitive changes should require reauthentication where appropriate.

Include tests.

5. Settings / Preferences / Customization

Missing:

Settings versioning

Preferences may evolve.

Store/version settings so migrations can safely transform:

v1 → v2 → v3

Sensitive setting protection

Changes to:

email

password

MFA

API credentials

billing

ownership

must not use the same trust level as changing a theme.

Preference precedence

Define:

system default

→ product default

→ organization setting

→ user setting

→ session-specific override

where applicable.

Invalid preference recovery

If a stored preference becomes invalid after a deployment:

validate it

fall back safely

migrate it

never crash the application.

Privacy preferences

Separate:

functional preferences

analytics consent

marketing consent

personalization.

Import/export compatibility

Settings exports should have:

schema version

validation

migration support.

06 — ADMIN DASHBOARD & OPERATIONS CONTROL

Build a completely independent admin dashboard that belongs to this product only.

This is NOT a central dashboard for multiple products.

It manages ONLY the current application's:

- users
- organizations
- subscriptions
- payments
- usage
- analytics
- errors
- jobs
- webhooks
- audit logs
- support tickets
- feature flags
- system settings

Design it to be extremely minimal.

Sidebar:

Overview
Users
Organizations
Billing
Usage
Jobs
Webhooks
Analytics
Security
Audit Logs
Support
Feature Flags
System
Settings

Dashboard cards:

Active Users
Revenue
Subscriptions
Errors
Usage
Failed Jobs
Webhook Failures

Admin security:

- separate admin authorization
- RBAC

- MFA
- session timeout
- audit every sensitive action
- confirmation for destructive actions
- prevent ordinary users accessing admin routes

Support safe user impersonation only when explicitly enabled, with prominent impersonation indicators and complete audit logs.

Never expose passwords, tokens, API keys or secrets.

6. Admin Dashboard & Operations Control

Missing:

Break-glass administration

Create a controlled emergency-admin mechanism with:
strong authentication

explicit reason

limited duration

automatic audit

elevated visibility.

Admin action confirmation

High-risk actions should require:

explicit confirmation

typed confirmation for destructive actions

reauthentication for critical operations.

Dangerous-action separation

Separate normal operational actions from destructive actions.

For example:

Suspend user

Delete user

Delete tenant

Refund payment

Change owner

Rotate API keys

should not be equivalent to ordinary CRUD.

Admin session security

Add:

short inactivity timeout

absolute lifetime

stronger authentication

session revocation

admin-specific session visibility.

Admin UI data minimization

Do not expose more personal/customer data than necessary.

Admin API separation

The admin frontend must not be trusted merely because it is an internal UI.

Every admin endpoint independently validates:

authentication
authorization
input
resource ownership/context
Bulk actions
Bulk operations require:
bounded batch size
partial-failure handling
transaction strategy
progress
resumability
audit records.
Dangerous query protection
Admin search/filter endpoints need:
query limits
pagination
timeout protection
expensive-query controls.

07 — FILE STORAGE & MEDIA PROCESSING

Build a reusable secure file-storage module.

Support:

S3-compatible storage
Cloudflare R2
UploadThing

Features:

- presigned uploads
- presigned downloads
- upload limits
- MIME validation
- extension validation
- filename sanitization
- image resizing
- thumbnails
- optimization
- deletion
- retention policies

Create:

```
<FileUpload />  
<ImageUpload />
```

<FilePicker />
<MediaPreview />

Never trust client MIME types.

Protect against:

path traversal
malicious files
oversized uploads
content-type spoofing
unauthorized downloads

Storage provider must be replaceable.

Large media processing must run asynchronously.

Content inspection

Do not trust:

filename
extension
Content-Type
alone.

Validate actual file content where practical.

Malware scanning

Architecture for asynchronous malware scanning.

Potential state:

UPLOADING
→ SCANNING
→ SAFE
→ REJECTED

Do not make untrusted uploads immediately downloadable if your threat model requires scanning.

Image security

Protect against:

decompression bombs
extremely large dimensions
malformed image files
metadata abuse
dangerous SVG content.

Filename handling

Never use user filename directly as filesystem/storage path.

Generate internal object IDs.

Object authorization

Every download should verify authorization server-side.

Never assume possession of an object URL equals authorization unless intentionally designed as a signed temporary URL.

Upload quotas

Support:

per-file limit

user limit

tenant limit

plan limit

aggregate storage limit.

Orphan cleanup

Handle:

database record created

upload fails

and:

upload succeeds

database transaction fails

with reconciliation cleanup.

Range requests / streaming

Large files should not necessarily be loaded entirely into application memory.

EXIF/privacy

Consider stripping:

GPS metadata

camera metadata

hidden metadata

where appropriate.

Include tests.

08 — ANALYTICS, EVENT TRACKING & LOGGING

Build a privacy-conscious analytics infrastructure.

Create:

trackEvent()

identifyUser()

trackPageView()

trackConversion()

Support:

PostHog

Mixpanel

Google Analytics

Create structured event schemas.

Every event should optionally contain:

userId
organizationId
eventName
timestamp
metadata

Never send:

passwords
tokens
API keys
private content
unnecessary sensitive information

Analytics failure must never break the application.

Also create structured application logging:

logger.info()
logger.warn()
logger.error()

Support:

request ID
environment
service
error code
timestamp

Event schema governance

Every event needs a version.

Example:

project.created.v2

rather than silently changing payload structure.

Event contract validation

Validate analytics events before ingestion.

Event deduplication

Prevent duplicate events from retries and page replays.

Event ordering

Do not assume client-generated events arrive chronologically.

Consent-aware tracking

Analytics collection must respect applicable privacy choices.

Data retention

Define retention per category:

analytics
application logs
security events

audit events.

Log correlation

Every request should ideally have:

requestId

traceId

userId if available

tenantId if available

without leaking sensitive values.

Log integrity

Security/audit logs should be protected against unauthorized modification/deletion.

High-cardinality protection

Prevent someone from accidentally generating millions of unique metric labels.

PII classification

Explicitly classify fields:

public

internal

personal

sensitive

secret

and define logging rules.

Sampling

High-volume logs/traces should support controlled sampling without destroying important security events.

Include log redaction.

09 — RATE LIMITING, SECURITY & API PROTECTION

Build centralized security middleware for this independent product.

Implement:

- rate limiting
- CORS
- CSRF where appropriate
- security headers
- request validation
- response validation
- request-size limits
- API authentication
- authorization
- abuse prevention
- brute-force prevention

Support Redis/Upstash where appropriate.

Create configurable rate limits for:

- login
- signup
- password reset
- API endpoints
- AI endpoints
- file uploads
- webhooks

Implement appropriate:

- CSP
- HSTS
- X-Content-Type-Options
- Referrer-Policy
- Permissions-Policy
- frame protection

Do not use unsafe wildcard configurations in production.

This one needs a major expansion.

Request normalization

Normalize:

- headers

- paths

- encoded characters

- Unicode

- content types

before security decisions where applicable.

Request smuggling protection

Infrastructure/proxy configuration should protect against HTTP request smuggling/desynchronization.

SSRF protection

For any server-side URL fetching:

- block private IP ranges

- block localhost

- block metadata endpoints

- validate DNS resolution

- prevent DNS rebinding

- restrict protocols

- restrict redirects.

ReDoS

Avoid unsafe regular expressions on user input.

Prototype pollution

Validate untrusted JSON/object structures carefully.

Resource exhaustion

Protect:

request body

JSON nesting

multipart fields

query complexity

pagination

regex processing

image processing

PDF processing.

Security headers

Explicitly test:

CSP

HSTS

X-Content-Type-Options

Referrer-Policy

Permissions-Policy

frame protections.

Dependency security

Automated:

dependency audit

lockfile validation

vulnerability alerts

abandoned-package detection.

Secret scanning

CI should scan for:

API keys

private keys

tokens

credentials.

API versioning

Define compatibility strategy:

/api/v1

/api/v2

rather than breaking existing consumers silently.

Request replay protection

For sensitive APIs:

timestamp

nonce

idempotency key

signature where appropriate.

Error normalization

Never expose:

stack traces

SQL

filesystem paths

provider credentials

internal service topology.

Document every security decision.

10 — AI WRAPPER & BACKGROUND JOB QUEUE

Build a provider-independent AI infrastructure.

Support:

OpenAI

Anthropic

Google

Create:

AIProvider

Functions:

generateText()

streamText()

generateStructured()

embed()

moderate()

Support:

- streaming
- retries
- timeouts
- cancellation
- model fallback
- token tracking
- cost tracking
- rate limiting
- usage limits
- structured output
- schema validation
- prompt versioning

Never allow the frontend to directly access provider API keys.

Add background jobs using:

Trigger.dev

BullMQ

or an equivalent durable system.

Support:

- retries
- backoff
- dead-letter jobs
- concurrency
- cancellation
- progress
- idempotency

Prompt injection defense

Treat external/user-provided content as untrusted.

Explicitly separate:

- system instructions
- developer instructions
- trusted application context
- untrusted user content
- retrieved content
- tool output

Tool-call authorization

An AI must never be allowed to perform an action merely because it generated a valid-looking tool call.

Every tool call needs normal server authorization.

AI output validation

Structured output must be:

generated

→ parsed

→ schema validated

→ business-rule validated

→ sanitized

→ executed

AI hallucination containment

Never allow AI output to directly become:

SQL

shell commands

authorization decisions

payment instructions

privileged actions.

Cost protection

Add:

per-user budget

per-tenant budget

per-feature budget

daily/monthly limits

emergency global kill switch.

Token accounting

Record:
model
input tokens
output tokens
cached tokens where applicable
estimated cost
actual provider cost if available.
Provider response normalization
Different AI providers fail differently.
Normalize:
timeout
rate limit
content policy refusal
context overflow
temporary provider failure
invalid response
Context limits
Before sending prompts:
calculate approximate token usage
truncate intelligently
summarize
reject impossible requests.
Job ownership
A queued job must preserve:
userId
tenantId
authorization context
requestId
jobId
but never blindly trust these values when executing.
Worker isolation
Long-running/expensive AI jobs should not block web requests.
Poison jobs
A job that repeatedly fails must eventually move to a DLQ instead of retrying forever.
Cancellation
Cancellation should actually stop downstream work where possible.
Provider outages
Support degraded behavior rather than infinite retries.

Protect against duplicate expensive AI jobs.

11 — ONBOARDING, PRODUCT TOURS & CHECKLISTS

Build reusable onboarding infrastructure.

Create:

<OnboardingChecklist />
<ProductTour />
<TooltipTour />
<WelcomeModal />

Support:

- ordered steps
- progress
- completion
- skipping
- resuming
- product-specific steps
- user-specific state

Example:

Create account
Complete profile
Create first project
Invite teammate
Complete first action

Persist state appropriately.

Do not rely solely on localStorage for important server-side progress.

Make onboarding configurable without rewriting components.

Onboarding state machine

Handle:

NOT_STARTED

IN_PROGRESS

COMPLETED

SKIPPED

RESET

Version onboarding

When onboarding changes:

onboarding_v1

onboarding_v2

users shouldn't unexpectedly lose progress.

Context-aware onboarding

Do not show onboarding for features the user cannot access.

Completion validation

Important onboarding completion must be confirmed server-side where it unlocks something meaningful.

Recovery

If onboarding gets interrupted halfway:

preserve progress

resume correctly

don't restart unnecessarily.

Accessibility

Tours need:

keyboard navigation

screen reader support

focus management

escape behavior.

Mobile behavior

Tour overlays need to handle:

viewport changes

scrolling

orientation

responsive layouts.

12 — FEATURE FLAGS & REMOTE CONFIGURATION

Build a secure feature-flag system.

API:

hasFeature()

Support targeting by:

user

organization

percentage rollout

environment

plan

role

Support:

- kill switches

- beta features

- gradual rollout

- disabled-by-default features

Feature flags must not become security authorization.

Authorization must always remain server-side.

Audit every flag modification.

Create a minimal admin UI for:

- create
- edit
- enable
- disable
- rollout
- delete

Support a simple database implementation with optional PostHog/
LaunchDarkly adapters.

Flag dependency graph

If:

feature A requires feature B

the system must handle dependencies explicitly.

Flag evaluation consistency

A user shouldn't randomly switch variants during a single request/session unless intentionally designed.

Configuration validation

Remote config should be schema validated before activation.

Safe defaults

If flag service is unavailable:

known safe default

must be used.

Flag expiration

Flags need:

owner

creation date

expiry/review date

lifecycle state.

Otherwise temporary flags become permanent technical debt.

Kill-switch safety

Kill switches should fail closed for dangerous functionality where appropriate.

Cache invalidation

Define propagation latency and stale-config behavior.

13 — AUDIT LOGS & ACTIVITY STREAM

Build an immutable audit-log system.

Record:

- actor
- action
- target

organization
timestamp
request ID
metadata

Examples:

LOGIN
LOGOUT
PASSWORD_CHANGED
PLAN_CHANGED
PROJECT_CREATED
PROJECT_DELETED
MEMBER_INVITED
API_KEY_CREATED
ADMIN_IMPERSONATION

Create:

logActivity()

Audit logs must:

- be append-only through normal application APIs
- not expose sensitive secrets
- support filtering
- support pagination
- support retention policies

Create an admin audit viewer.

Sensitive administrative actions must automatically generate audit events.

Tamper resistance

Application admins should not casually be able to rewrite historical audit events.

Audit event integrity

Consider:

append-only storage

restricted deletion

integrity mechanisms

separate retention controls.

Before/after values

For sensitive configuration changes, record safe before/after representations.

Never store secrets themselves.

Actor types

Support:

USER

ADMIN

SYSTEM
WORKER
API_KEY
WEBHOOK
AI_AGENT

Delegation

If an admin performs an action on behalf of a user:

actor = admin

subject = user

reason = ...

Do not misrepresent it as the user's action.

Audit reliability

Critical audit logging should not silently disappear because the primary application request failed.

Clock consistency

Use server-generated timestamps and consistent timezone handling.

14 — CUSTOMER SUPPORT, LIVE CHAT & FEEDBACK

Build reusable customer-support infrastructure.

Components:

<FeedbackWidget />

<BugReport />

<FeatureRequest />

<ContactSupport />

Support providers through adapters:

Crisp

Intercom

Plain

future providers

Capture:

user

organization

page

browser metadata where appropriate

description

screenshots optionally

Never accidentally expose private application data.

Allow administrators to view feedback from the product's own admin panel.

Include spam/rate limiting protection.

Authorization boundary

Support staff must only see information they're permitted to see.

Support impersonation

If implemented:

explicit user notification

time-limited session

restricted permissions

full audit

obvious impersonation indicator

no access to credentials/secrets.

Attachment security

Support uploads need the same security model as normal storage.

Conversation retention

Define:

retention

deletion

export

legal hold where applicable.

Spam prevention

Add:

per-user limits

IP limits

attachment limits

content-size limits.

Support identity verification

Before exposing account-sensitive information, support tooling needs a verification workflow.

SLA tracking

For serious support systems:

created

→ assigned

→ first response

→ pending customer

→ resolved

→ reopened

Duplicate ticket detection

Avoid creating dozens of tickets for the same event.

15 — SEO, OPEN GRAPH & META ENGINE

Build reusable SEO infrastructure for this independent product.

Support:

- title
- description
- canonical
- robots
- sitemap
- structured data
- OpenGraph
- social preview images
- dynamic metadata

Create:

```
generateMetadata()
generateSitemap()
generateStructuredData()
```

Ensure private application routes are not indexed.

Support product-specific SEO defaults.

Prevent duplicate canonical URLs.

Include validation and SEO tests.

SEO security

Prevent accidental indexing of:

dashboards

user pages

private workspaces

admin pages

internal APIs

temporary URLs.

Tenant SEO isolation

Custom-domain tenants must not accidentally expose another tenant's metadata.

Host-header poisoning

Never build canonical URLs blindly from arbitrary Host headers.

Sitemap scalability

Large products need:

pagination/sharding

incremental generation

caching

invalidation.

Structured-data validation

Schema.org output should be generated from validated data.

Locale SEO

Support:

hreflang

localized canonical URLs
language-region combinations.
Preview security
OG image generation should not become an SSRF/vector through arbitrary user-controlled URLs.
SEO deployment safety
Preview/staging environments should have explicit no-index behavior.

16 — LEGAL & COMPLIANCE LAYER

Build reusable legal infrastructure.

Create configurable pages for:

Terms of Service
Privacy Policy
Cookie Policy
Acceptable Use Policy
Refund Policy

Implement:

- consent tracking
- terms acceptance
- privacy preferences
- data export request
- account deletion
- retention policy infrastructure

Create:

```
<CookieConsent />  
<PrivacySettings />  
<TermsAcceptance />
```

IMPORTANT:

Do not pretend generic templates guarantee legal compliance.

Make jurisdiction-specific configuration possible.

Clearly document areas requiring review by qualified legal counsel.

Ensure user data deletion and export workflows are asynchronous where necessary and properly audited.

Policy versioning

Record:
terms_version
privacy_version
accepted_at
accepted_ip where legally appropriate
rather than merely:
accepted = true
Consent separation
Do not combine:
Terms acceptance
Privacy acknowledgement
Marketing consent
Cookie consent
into one boolean.
Consent withdrawal
Users need a mechanism to withdraw optional consent.
Data subject workflow
Support state machines for requests such as:
ACCESS
EXPORT
CORRECTION
DELETION
RESTRICTION
where applicable.
Legal hold
Deletion systems must be able to respect legitimate retention/legal-hold requirements.
Retention engine
Data categories should have configurable retention policies.
Vendor/data-processing inventory
Maintain an internal list of:
processors
data categories
purposes
regions.
Policy localization
Legal content may vary by jurisdiction.
Do not assume one global policy text fits every market.
Consent evidence integrity
Consent records should be protected against ordinary user/admin modification.

17 — GLOBAL SEARCH & COMMAND PALETTE

Build a reusable Cmd/Ctrl + K system.

Support:

- navigation
- database search
- workspace switching
- creating resources
- opening settings
- running quick actions

Commands must be:

- permission-aware
- context-aware
- keyboard accessible

Create:

<CommandPalette />

Allow each independent product to register its own commands.

Do not make the command system dependent on another application.

Prevent unauthorized actions from appearing or executing.

Search authorization

Search results must be permission-filtered before returning results, not merely hidden in UI.

Search index security

Removing access to a document should eventually remove/disable it in search.

Search leakage

Prevent snippets from exposing:

- private content
- deleted content
- hidden metadata.

Query abuse

Protect against:

- enormous queries
- expensive wildcard searches
- pathological search expressions.

Search consistency

Define expected behavior when:

- database updated
- but search index hasn't updated yet.

Reindexing

Need:

- full reindex
- incremental reindex
- failed-document retry
- progress
- index versioning.

Permission changes

When permissions change, cached search results must not remain accessible indefinitely.

Command safety

Destructive command-palette actions need confirmation.

18 — WEBHOOK ENGINE & PUBLIC API KEY MANAGEMENT

Build reusable public API infrastructure.

API keys must:

- be generated securely
- be hashed at rest
- only be shown once
- support scopes
- support expiration
- support revocation
- track last usage
- support rate limits

Webhook system:

- endpoint registration
- event subscriptions
- HMAC signatures
- delivery retries
- exponential backoff
- delivery logs
- replay
- failure handling
- dead-letter state

Prevent:

replay attacks

duplicate events

signature bypass

unauthorized endpoint access

Never store plaintext API keys.

Key lifecycle

Support:

CREATED

ACTIVE

ROTATED

REVOKED

EXPIRED

Key prefix identification

Store a non-secret prefix for identifying keys without exposing the secret.

Key rotation

Allow overlapping old/new keys for controlled migration.

Scope hierarchy

Prevent a key from granting permissions greater than the creator is allowed to grant.

Webhook secret rotation

Support old/new secrets during rotation.

Destination validation

Protect webhook configuration against SSRF.

Webhook endpoint ownership

Users should prove ownership/control of destinations where appropriate.

Delivery isolation

One failing webhook destination must not block other destinations.

Payload versioning

Webhook schemas need versions.

Consumer acknowledgement

Clearly define what HTTP responses mean:

2xx

4xx

5xx

timeout

Replay controls

Replay must preserve security semantics and avoid triggering irreversible actions twice.

API pagination contracts

Define:

cursor format

cursor expiration

stable ordering

duplicate avoidance

deleted-record behavior.

API compatibility

Document deprecation periods and compatibility guarantees.

19 — EXPORT, IMPORT & DATA PORTABILITY

Build reusable data portability infrastructure.

Support:

CSV

JSON

PDF where appropriate

Import pipeline:

- upload
- parse
- validate
- preview
- duplicate detection
- transaction
- rollback
- error reporting

Large imports/exports must use background jobs.

Export must respect authorization and tenant boundaries.

Never allow a user to export another user's data.

Provide progress tracking.

Create reusable:

- <ExportButton />
- <ImportWizard />
- <ExportProgress />

Export consistency

A large export should represent a coherent snapshot rather than randomly mixing data from different moments.

Export authorization

The authorization check must occur when the export is requested and when sensitive download access is granted.

Export encryption

Highly sensitive exports should have appropriate protection at rest and during transfer.

Export expiration

Generated export files should automatically expire/delete.

Import isolation

Imported files should be processed in isolated temporary storage.

Import validation

Validate:

- schema
- relationships
- references
- enums
- IDs
- duplicate records
- maximum rows.
- Import dry run
- Provide:
 - validate only

before committing large imports.

Partial failure strategy

Clearly choose:

all-or-nothing

or controlled partial import.

Import rollback

Large imports need rollback/recovery strategy.

Malicious imported content

Imported data must be treated as untrusted even if it came from another product instance.

20 — AFFILIATE & REFERRAL SYSTEM

Build a reusable referral system.

Support:

referral codes

referral links

cookie attribution

signup attribution

conversion tracking

commission ledger

fraud detection

payout status

Example:

?ref=ABC123

Referral attribution must be securely validated.

Prevent:

self-referrals

duplicate attribution

tampering

fake conversions

Integrate with billing only through clean interfaces.

Support future providers such as:

Rewardful

FirstPromoter

Create admin reporting.

Attribution model

Define exactly:

first-touch

last-touch

signup-time

conversion-time

coupon attribution.

Attribution expiry

Referral attribution should expire predictably.

Cookie security

Referral cookies require:

secure configuration

expiration

tamper resistance

correct domain/path.

Commission state machine

PENDING

→ APPROVED

→ PAYABLE

→ PAID

with:

REVERSED

DISPUTED

where necessary.

Refund handling

A refunded transaction should correctly reverse associated commission.

Affiliate account security

Affiliate payout details are sensitive.

Require stronger authentication for changing them.

Fraud review

Support manual review rather than blindly paying suspicious referrals.

Self-referral detection

Use multiple signals rather than a single client-controlled field.

Attribution tampering

Never accept:

affiliateId

from the client as authoritative without server verification.

21 — USAGE-BASED METERING & CREDIT SYSTEM

Build a transaction-safe usage and credit ledger.

Track:

AI tokens

API calls

video exports
storage
processing time
credits

Create:

recordUsage()
getUsage()
deductCredits()
refundCredits()
checkLimit()

Every transaction must be:

atomic
idempotent
auditable

Prevent:

double charging
negative balances
race conditions
duplicate webhook credits

Support:

free limits
subscription limits
hard limits
soft limits
credit purchases
monthly resets

Create usage dashboard.

Ledger architecture

Do not only store:

credits = 400

Use an immutable transaction ledger where practical:

+100 grant

-20 usage

+50 refund

-10 correction

Balance reconciliation

Periodically calculate:

ledger total

vs

materialized balance

Negative balance protection

Atomic transactions must prevent concurrent requests from spending the same credits.

Reservation model

For expensive operations:

reserve

→ execute

→ settle

instead of immediately consuming the entire amount.

Failed operation refund

If the operation fails after reservation:

reservation → released/refunded

Partial usage

Support operations that consume an unknown amount until completion.

Credit expiry

If credits expire:

expiration date

priority consumption

notification

ledger event.

Pricing versioning

Usage must retain the pricing version under which it was calculated.

Meter correction

Admin corrections must be:

audited

reason-required

non-destructive to original records.

Concurrent usage

Two simultaneous requests must not both pass:

remaining $\geq$ cost

using stale balances.

22 — MULTI-LANGUAGE & I18N

Build reusable internationalization infrastructure.

Support:

multiple languages

dynamic language switching

currency formatting

date formatting

timezone formatting

pluralization

RTL layouts

Use:

next-intl
react-i18next
or an equivalent maintained library.

Create typed translation keys where practical.

Do not hard-code user-facing strings throughout components.

Support product-specific translation files.

Ensure missing translations fail gracefully.

Translation fallback hierarchy

For example:

fr-CA

→ fr

→ default language

Translation versioning

When keys change, detect stale translations.

Missing translation detection

CI should fail/warn on missing keys.

Runtime translation safety

Never execute translation strings as HTML/scripts without sanitization.

User-generated content

Separate translated UI strings from user content.

Number formatting

Support locale-specific:

decimal separators

grouping

percentages

currencies.

Calendar systems

If relevant, don't assume every locale uses identical calendar conventions.

Timezone/DST

Store canonical timestamps and convert at presentation boundaries.

RTL testing

Test layout, not just translated strings.

Text expansion

UI must survive languages where text is substantially longer.

23 — MAINTENANCE MODE, STATUS & SYSTEM BANNERS

Build a system-status module for this independent product.

Support:

- maintenance mode
- scheduled maintenance
- incident banners
- warning banners
- service degradation
- breaking updates

Create:

<SystemBanner />

Configuration should support:

- global
- authenticated users
- specific routes

Maintenance mode must provide an admin bypass.

Do not accidentally lock administrators out.

Include caching and graceful fallback.

Emergency access

Maintenance should have a secure emergency/admin bypass that cannot be activated by a client-controlled flag.

Maintenance scope

Support:

- whole system

- tenant

- route

- feature

API version.

Read-only mode

Sometimes you don't want full downtime.

Support:

READ_ONLY

where appropriate.

Maintenance consistency

Background workers should also understand maintenance state.

API behavior

Don't return an HTML maintenance page to API clients expecting JSON.

Scheduled maintenance timezone

Store schedules with explicit timezone/UTC semantics.

Status communication

Separate:

internal incident state

from:

public status message.

Recovery validation

Don't immediately mark the system healthy after deployment; verify dependencies first.

24 — CHANGELOG, ROADMAP & WHAT'S NEW

Build reusable product update infrastructure.

Support:

public changelog

roadmap

release notes

What's New modal

unread tracking

Create:

<WhatsNew />

<ChangelogBadge />

<Roadmap />

Use markdown/MDX or a database-backed system.

Allow admin users to:

create

edit

publish

schedule

archive

Track which users have viewed updates.

Keep this module independent from external products.

Publication workflow

Support:

DRAFT

→ REVIEW

→ SCHEDULED

→ PUBLISHED

→ ARCHIVED

Content security

Markdown/MDX content must be sanitized and restricted.

Admin content permissions

Not every admin should automatically be able to publish public content.

Version targeting

What's New can be targeted based on:

product version

feature availability

user segment.

Read-state synchronization

Unread status must handle multiple devices.

Release correlation

Changelog entries can optionally associate with:

release/version

feature flag

deployment

Broken-content prevention

Validate links/assets before publication.

25 — CUSTOM DOMAIN & WHITE-LABEL ENGINE

Build a reusable custom-domain infrastructure.

Support:

custom domains

domain verification

DNS instructions

custom branding

logos

colors

favicon

Potential infrastructure:

Cloudflare

Vercel

reverse proxy

Architecture must prevent:

host-header attacks

tenant confusion

domain spoofing

unauthorized domain claiming

Require domain ownership verification.

Never trust a hostname as proof of ownership.

Create domain management UI.

SSL provisioning should be abstracted behind an infrastructure provider.

Domain ownership verification

Verification must prevent:

DNS spoofing assumptions

verification replay

stale verification

domain takeover.

Domain lifecycle

REQUESTED

→ VERIFYING

→ VERIFIED

→ ACTIVE

→ FAILED

→ DISABLED

DNS rebinding

Do not assume a domain that was verified once remains controlled forever without considering revalidation/security implications.

Host routing isolation

Request:

customer-a.example.com

must never resolve to customer B due to cache/routing bugs.

Cache isolation

CDN/application caches must include host/tenant context.

Certificate lifecycle

Handle:

issuance

renewal

expiration

failed renewal

revoked certificate.

Domain removal

When a domain is removed:

invalidate routing

revoke/disable association

purge relevant caches.

White-label asset isolation

Customer A must never be able to request customer B's branding assets.

Email branding

If white-labeling emails, ensure the sending domain is properly configured rather than spoofing arbitrary From addresses.

DNS instructions

Don't blindly tell users to create dangerous DNS records without explaining exact expected values.

26 — PRODUCT HEALTH, MONITORING & INCIDENT SYSTEM

Build a self-contained application health and incident monitoring system.

Monitor:

application uptime

API latency

error rate

database connectivity

queue health

external provider health

background jobs

webhooks

Create health endpoints:

/health

/ready

/live

Differentiate:

liveness

readiness

dependency health

Create internal admin dashboard:

System Health

Services

Errors

Latency

Incidents

Maintenance

Support incident states:

investigating

identified

monitoring

resolved

Create optional public status page.

Do not expose sensitive infrastructure details publicly.

Integrate with:

Sentry

OpenTelemetry

provider health APIs

Provide graceful degradation when monitoring itself fails.

SLO/SLA model

Define measurable objectives such as:

availability

latency

error rate

job success rate

rather than only showing dashboards.

Alert deduplication

Prevent one incident from generating 5,000 alerts.

Alert grouping

Group related failures into one incident.

Dependency-aware health

Distinguish:

application healthy

database unhealthy

payment provider degraded

AI provider unavailable

rather than one generic red status.

Synthetic monitoring

Perform real user-flow checks:

signup

login

payment

critical product action

where appropriate.

Incident lifecycle

DETECTED

→ ACKNOWLEDGED

→ INVESTIGATING

→ MITIGATED

→ RESOLVED

→ POSTMORTEM

Alert fatigue controls

Support:

severity

cooldown

suppression

maintenance windows.

Runbooks

Each critical alert should link to a remediation/runbook.

Postmortems

Store:

impact

timeline

root causes

contributing factors

remediation.

Monitoring privacy

Never put credentials, tokens, or private customer data into monitoring payloads.

27 — ENVIRONMENT & DEPLOYMENT MANAGEMENT

Build reusable environment and deployment infrastructure for ONE independent product.

Support environments:

development

preview

staging

production

Create:

typed environment validation

startup configuration validation

environment-specific configuration

feature configuration

deployment metadata

Never commit secrets.

Never expose server secrets to client code.

Create deployment documentation for:

Vercel

Cloudflare

Docker

generic VPS

Support:

deployment version
commit SHA
build timestamp
environment

Create an admin/system page showing:

Version
Environment
Deployment time
Git commit
Runtime
Database version

Never expose secrets through this dashboard.

Include deployment rollback documentation.

Deployment gates

Production deployment should optionally require:

tests

- typecheck
- build
- migration check
- security checks
- approval

Migration/deployment ordering

Handle:

old application

+ new database

compatibility before deploying breaking code.

Zero-downtime migrations

Avoid:

deploy code

- database column suddenly disappears

Use expand/contract migration patterns.

Config drift detection

Detect differences between expected and actual environment configuration.

Secret rotation

Support rotation without requiring dangerous downtime.

Deployment rollback

Rollback strategy must account for database migrations.

Artifact integrity

Deploy immutable build artifacts rather than rebuilding different code during rollback.

Canary deployment

For higher-scale products, support gradual deployment.

Health-based rollback

Automatically halt/rollback when deployment health deteriorates where safe.

Build reproducibility

Lock dependencies and build configuration sufficiently to reproduce releases.

Source-control protection

Production deployment should not depend solely on a developer's local machine.

28 — DATABASE MIGRATION, BACKUP & DISASTER RECOVERY

Build reusable database reliability infrastructure.

Support:

schema migrations

version tracking

seed data

database backup procedures

restore procedures

migration verification

rollback strategy

Database must use:

foreign keys

indexes

constraints

transactions

Create migration safety checks.

Prevent destructive migrations from running accidentally in production.

Create scripts for:

db:migrate

db:rollback where safely possible

db:seed

db:backup

db:verify

Document:

RPO

RTO

backup retention

restore testing

disaster recovery

IMPORTANT:

A backup must not be considered reliable merely because it was created.

Provide a documented restore verification procedure.

Never allow production database credentials into client code.

Point-in-time recovery

Backups alone aren't enough.

Support, where infrastructure permits:

restore database

to specific point in time

Backup encryption

Backups should be encrypted and access-controlled.

Backup isolation

A compromised production account should not automatically be able to destroy every backup.

Immutable backups

Use retention/immutability mechanisms where appropriate.

Restore drills

Automated or scheduled restore tests.

Backup monitoring

Alert when:

backup fails

backup becomes too old

backup size changes unexpectedly

restore verification fails.

Migration compatibility

Migrations should be tested against realistic production-like datasets.

Migration locking

Prevent two deployment processes from simultaneously executing migrations.

Destructive migration protection

Require additional safeguards before:

dropping tables

dropping columns

destructive transformations.

Data integrity checks

After migrations:

row counts

foreign keys

critical invariants

application smoke tests.

Disaster recovery documentation

Explicitly document:

RPO

RTO

- restore procedure
- credentials required
- dependencies
- owner
- verification

29 — EXPERIMENTATION & A/B TESTING

Build reusable experimentation infrastructure.

Support:

- experiments
- variants
- user allocation
- percentage allocation
- organization allocation
- conversion tracking
- experiment lifecycle

Create:

- `createExperiment()`
- `getVariant()`
- `trackConversion()`

Assignments must be deterministic where appropriate.

A user should not randomly jump between variants.

Support:

- draft
- running
- paused
- completed

Create minimal admin interface:

- Experiment
- Variants
- Traffic
- Conversions
- Results

Do not automatically declare a statistical winner.

Show raw metrics and methodology.

Experiments must never override security permissions.

Do not use experimentation to determine authorization.

Experiment eligibility

Users should only enter experiments if they satisfy defined eligibility criteria.

Assignment persistence

Once assigned:

control

shouldn't unexpectedly become:

variant

unless intentionally designed.

Exposure tracking

Separate:

assigned

from:

actually exposed

because a user may be assigned but never see the feature.

Experiment contamination

Prevent users from simultaneously entering incompatible experiments.

Mutual exclusion

Support experiment groups.

Sample-ratio monitoring

Detect unexpected assignment imbalance.

Metric definitions

Each experiment needs:

primary metric

secondary metrics

denominator

attribution window

exclusion criteria.

Data-quality checks

Detect:

missing events

duplicate events

bots

tracking failures.

Experiment lifecycle

DRAFT

→ RUNNING

→ PAUSED

→ COMPLETED

→ ARCHIVED

Experiment cleanup

When an experiment ends:

remove dead code

remove flags

preserve historical data.

Privacy

Don't expose experiment assignment unnecessarily to clients where it can affect sensitive behavior.

30 — SECURITY CENTER / VULNERABILITY & COMPLIANCE DASHBOARD

Build a reusable security center for this independent product.

Create a security dashboard showing:

authentication status

MFA adoption

active sessions

API keys

recent security events

failed login attempts

rate-limit events

webhook failures

dependency vulnerabilities

configuration warnings

Create automated security checks for:

missing security headers

unsafe environment variables

outdated dependencies where detectable

weak configuration

publicly exposed routes

missing authorization checks

dangerous CORS

insecure cookies

debug mode in production

Create:

SecurityEvents

SecurityAlerts

SecurityChecks

Support severity levels:

LOW

MEDIUM
HIGH
CRITICAL

Do not expose sensitive security findings to ordinary users.

Admin-only access.

Create remediation guidance for each finding.

Never claim regulatory certification automatically.

The system should identify technical configuration issues, not falsely declare legal compliance.

Security posture history

Don't only show:

CURRENT: 7 warnings

Store historical security posture.

Finding lifecycle

OPEN

→ ACKNOWLEDGED

→ IN_PROGRESS

→ MITIGATED

→ VERIFIED

→ CLOSED

False-positive handling

Allow documented suppression with:

reason

owner

expiration

approval.

Security finding deduplication

The same vulnerability shouldn't generate thousands of findings.

Dependency vulnerability prioritization

Distinguish:

vulnerable dependency exists

vulnerable code path is actually reachable

exploitability

severity

exposure.

Secret exposure response

If a secret is discovered:

detect

→ alert

→ revoke

→ rotate

→ investigate

not merely “show warning.”

Security incident correlation

Correlate:

failed logins

+ unusual API calls

+ key usage

+ admin action

when appropriate.

Security baseline

Define a machine-readable baseline:

required headers

required cookie settings

required auth controls

required encryption

required logging

Security regression testing

Run security checks in CI/CD so a secure configuration doesn't silently regress.

Dependency SBOM

Generate software-bill-of-materials information for releases where appropriate.

Supply-chain security

Consider:

dependency lockfile integrity

trusted registries

signed releases/artifacts

CI permissions minimization

compromised-package detection.

Security notifications

Notify appropriate administrators when:

new privileged login

API key created

API key rotated

MFA disabled

password changed

email changed

owner changed

suspicious activity detected.

Add security regression tests.