The Company

DevNest is a fast-growing, cloud-native SaaS service-oriented company that grew its business by launching high-demand products quickly and then standardizing shared infrastructure later. The business model for DevNest is to respond to market demand for new products faster than anyone else, thereby establishing market share while the competition is still deciding whether to enter the market.

DevNest relies on an in-house market research team that uses real-time data, advanced data analytics, and AI machine learning models to predict future trends and market demand. This team operates as a separate organization and is not connected to or involved with any operational divisions.

The company is focused on selling enterprise bundles, but it still feels like a “startup with enterprise customers.” At this point, the company's long-term business strategy is to develop new business areas, identify those that have long-term growth and revenue potential, and then focus on building more well-defined, disciplined operations to sustain long-term revenue streams.

Structure

DevNest operates four distinct business lines, each with some autonomy. They all rely on a shared IT environment, with each line of business responsible for its own applications, which it uses exclusively.

However, all the specialized applications sit on top of shared internal utilities, which are treated like IT plumbing. Everyone depends on them, so each team contributes part of its own IT resources to support the shared utilities. IT operations is led by a small engineering team, headed by Dmitri Chernikov, the operations center senior manager.

Corporate Strategy

DevNest understands that they can't go toe to toe with the large tech giants, the established players, and deep pockets and economies of scale that make it difficult to compete on their terms.

Instead, DevNest has decided to beat them by being more agile and responsive to emerging trends and market conditions than IBM, Meta, Oracle, and the others. The very size of these giants makes it difficult for them to respond quickly, and the entrenchment of their mature processes in their corporate culture also means the costs of entering a new, small, or niche market are not worth the risks.

The basic strategy of DevNest is to identify emerging markets and trends, establish a leadership position in those markets, and, once the market position is established, focus on product quality, risk, SLAs, KPIs, and other metrics to ensure a stable customer base.

Risk Culture

New Development Risk

DevNest's business growth depends on entering new areas of business where operational risks are still relatively unknown. These new companies are often based on emerging technologies. DevNest has achieved most of its rapid business growth by being a leader in adopting new business ideas and innovative technologies to establish a leadership position for its products or services in the marketplace.

From a risk perspective, management expects a majority of these ventures to fail. Still, they accept that as the cost of finding those few products or services with the potential to deliver explosive growth and returns on investment.

This requires the organization to be agile, to adopt new ideas, and to fail often and early as they explore the potential of a new business. Because of the high failure rate of these ventures, only minimal basic risk mitigation is implemented to avoid increasing time-to-market or costs. This agile approach, based on lean development, holds that because there is little historical data on risk, the best way to understand it is through trial and error.

This approach emphasizes CI/CD, Agile methods, and just enough development to deliver working prototypes to the marketplace. Risk management is not a high priority beyond standard industry common risks.

There is little or no external governance for these new products since they are often in production before any official standards or oversight are established.

The view championed by the Director of Business Development advocates a very Agile approach, emphasizing risk avoidance and acceptance as products are being developed.

Mature Product Risk

Once a product shows a sustained, growing revenue stream, it is upgraded to supported product status.

These revenue-generating products and services gradually become dominated by customers who require SLAs, audit evidence, and predictable change control. Some of the management team are pushing hard for robust IT risk maturity development because outages and weak governance are starting to threaten renewals and customer loyalty.

Line of Business Descriptions

CommerceHub (Commerce Line)

CommerceHub is DevNest's flagship product: a payments + checkout + settlement platform for mid-market retailers and subscription businesses. Customers embed its APIs/SDKs into their sites or POS systems to:

- create carts and orders
- price products with promos/taxes
- take payments and authorize cards
- manage refunds/chargebacks
- settle funds to merchants daily

The customer base is composed of:

- Retailers (5k–20k monthly orders each)
- Subscription platforms
- Some "white-label" banks and marketplaces

Business criticality

This division carries the highest direct revenue risk. Every minute of downtime during peak retail windows = immediate merchant loss + SLA penalties.

The system's load is seasonal. Black Friday/holiday spikes create operational pressure and emergency work, as the added stress on systems can cause failures not seen at more normal levels of operation.

Typical architecture

- Public Checkout API (REST + GraphQL) behind edge and web application firewalls
- Order Management microservices (Kubernetes / ECS)
- Payment workflow engine (event-driven, high consistency)
- Settlement / ledger service (strict audit trail)
- Fraud screening hooks (calls to Analytics + third-party vendors)

Key dependencies

- IdentityCore for merchant/admin logins, token validation
- Shared NoSQL for carts, orders, settlement states
- Shared DNS/service discovery for internal routing
- Cloudflare-like edge for bot protection (checkout attacks are common)

Operational reality

- Deployment pace is high.
- Commerce teams push config changes multiple times per day.
- Emergency changes are normalized during peaks.
- Retry logic is aggressive by design, so upstream failures can cause storms.

IdentityCore (Identity Line)

IdentityCore provides authentication and authorization for every DevNest customer-facing product. It is both a standalone SaaS identity product and an internal platform utility. Capabilities include:

- Single sign on for enterprise tenants
- JWT token issuing + validation
- MFA + device posture checks
- Session management
- Directory sync (SCIM)
- Admin RBAC policy model

Used by:

- Every DevNest product
- Customers who buy IdentityCore as a standalone IAM solution

Business criticality

If IdentityCore fails, customers cannot log into any DevNest system, and internal services may fail token checks.

It is a classic “shared service with fuzzy ownership,” meaning a platform team runs it, but product teams treat it as “just there.” Like plumbing in an office building.

Typical architecture

- Auth Gateway (rate-limited, edge terminated)
- Token Service (stateless microservices + signing keys in KMS/HSM)
- Policy Engine (low-latency authorization decisions)
- User directory + sessions stored in NoSQL w/TTL
- MFA & device services plus 3rd-party integrations (SMS/email, push)

Key dependencies

- Shared IAM/PAM (for internal operator access)
- Shared NoSQL (sessions, users, policies, audit entries)
- Observability (high-signal KRI/KPIs needed here due to systemic role)

Operational reality

- Must be “boringly stable,” but inherits the same CI/CD and DNS tooling as the rest of DevNest.
- Changes are infrequent but have a high blast radius.
- Ownership for KRIs/residual risk is often disputed because it straddles “platform and product.”

InsightStream (Analytics Line)

InsightStream is a cross-product analytics and event intelligence platform. It collects events from all DevNest products and provides:

- real-time merchant dashboards
- anomaly detection for fraud/spikes
- internal product telemetry
- data feeds for AI risk scoring models

Used by

- DevNest internal teams (product, risk, fraud, exec dashboards)
- Partners consuming event streams via API

Business criticality

Does not directly block customer access but:

- fraud models degrade without it
- executive visibility collapses
- regulatory reporting “data integrity” risk rises

During incidents, InsightStream often fails early because it depends on shared storage and DNS.

Architecture

- Event collectors/SDKs in apps (Kafka/Kinesis-like ingestion)
- Stream processing (Flink / Spark Structured Streaming)
- Feature store/aggregation service
- Analytics APIs + dashboards
- Cold storage lake (S3-like) + warehouse (Snowflake/Redshift-like)

Key dependencies

- Shared NoSQL for near-real-time counters and state
- Shared CI/CD for pipeline configs
- Shared observability (but its own pipelines also feed observability)

Operational reality

- Pipeline configs are brittle; schema drift is common.
- Backpressure or retries during outages can multiply the load on NoSQL.
- Often treated as a “nice to have” until a crisis shows its importance.

IntegrationsEdge (Integrations Line)

IntegrationsEdge is DevNest’s public API + partner ecosystem layer. It provides:

- Partner APIs for order, identity, analytics access
- Webhooks and event callbacks
- Prebuilt connectors to 3rd parties (ERP, CRM, banks, shipping services)
- Tenant-level throttling and SLA enforcement

Used by

- Enterprise partners embedding DevNest in their own platforms
- Systems integrators and MSPs
- Banks/marketplaces reselling DevNest services

Business criticality

- High contractual and reputational risk.
- Enterprise contracts have explicit latency/availability clauses.
- Issues can trigger partner churn quickly.

Typical architecture

- API gateway layer (rate-limited, per-tenant policy)
- Webhook dispatch system
- Connector microservices (often vendor-specific)
- Schema translation & mapping layer
- Partner sandbox + key management

Key dependencies

- IdentityCore for partner auth and token introspection
- Shared edge for traffic shaping + bot defense
- Shared DNS/service discovery
- Shared NoSQL for API keys, webhook state, and idempotency keys

Operational reality

- Vendor footprint is large and unevenly governed.
- Many integrations were built quickly and never risk-reviewed.
- Partner traffic spikes are unpredictable; retries can be brutal.

Shared Technical Infrastructure

Note that not all of the information presented here will be useful to you. Part of the challenge of the capstone is to sift through the data and only retain the facts that are useful for your analysis. Use this section as a reality check on some of your analyses.

Shared IAM & Privileged Access (PAM)

A centralized identity plane for DevNest employees and automation that manages:

- workforce SSO
- role-based admin access
- break-glass accounts (emergency-use-only privileged accounts that let you regain access to critical systems when typical access paths fail)
- service accounts for managing CI/CD and infrastructure tools
- approval workflows for privileged ops accounts (in theory)

Uses an enterprise security stack:

- central employee identity & SSO (IAM),
- tight control of admin credentials (PAM),
- strong cryptographic key management (KMS/HSM),
- least-privilege, time-bounded admin access (JIT/JEA).

Real-world posture

- Privileged access sprawl happened during growth. Many engineers still retain broad roles.
- Peer review for changes may be policy, but it is not enforced as a guardrail.
- “Emergency override” paths exist for speed and are used often.

Centralized DNS & Service Discovery

The internal routing control plane supporting:

- sservice-to-service discovery (how one backend service finds and connects to another without hard-coding IPs/hosts)
- internal DNS records for microservices
- health-based routing for failover - Traffic is routed based on real-time health checks.
- config pushed through automation instead of being done manually

Tech Stack

- AWS Route53 private hosted zones / Consul / etcd-backed registry
- Sidecars or service mesh (Envoy/Istio-like), but uneven adoption
- Automated DNS refresh tools

Real-world posture

- DNS “automation” has a global scope (DNS automation is a powerful, shared lever, but one move can change routing/discovery across the whole organization)
- Dependency tagging hygiene is inconsistent (the discovery system’s “map” of services is messy because the labels feeding it are messy)
- Small mistakes can poison discovery for core services (a small human or automation error can break how critical services are located, causing widespread outages)

Global Edge + Bot Protection (Cloudflare-like)

Front door for all customer and partner traffic supporting

- CDN caching
- WAF rules
- DDoS protection
- bot score/challenge
- geo routing / TLS termination

Tech Stack

- Cloudflare / Akamai / Fastly style edge
- central ruleset management
- auto-tuning bot mitigation
- per-tenant ACLs

Real-world posture

- The central team owns baseline configs; product teams request changes via tickets.
- Auto-tuning creates config churn, meaning lots of small changes make it hard to track normal baselines over time
- Edge is often a hidden systemic dependency until it fails.
- No transparent bypass mode for “edge hard-fail.” Specifically, if it goes down, there is no workaround in place.

Core NoSQL Datastore (DynamoDB-like)

The single most important data dependency that tracks:

- identity sessions
- carts/orders
- webhook state
- near-real-time analytics counters
- policy and config state

Tech Stack

- DynamoDB / CosmosDB / Cassandra-like managed NoSQL
- global tables / multi-region replication
- Time-to-live for sessions/events
- autoscaling + throughput alarms

Real-world posture

- The central team owns baseline configs; product teams request changes via tickets.
- Auto-tuning creates config churn, meaning lots of small changes make it hard to track normal baselines over time.
- Edge is often a hidden systemic dependency until it fails.
- No transparent bypass mode for “edge hard-fail.” Specifically, if it goes down, there is no workaround in place

CI/CD Automation for Config Rollouts

Shared build/deploy + infrastructure-as-code platform:

- app deployments
- feature flag rollouts (canary, blue/green, etc)
- DNS/edge config pushes
- Terraform/CloudFormation changes

Tech Stack

- GitLab/GitHub Actions/Jenkins + ArgoCD/Spinnaker
- IaC pipelines
- canary and blue/green support plans
- centralized “golden path” templates (runbooks)

Real-world posture

- Global automation blast radius is not always constrained. (A blast Radius is how significant the impact is if something goes wrong.)
- Some pipelines skip approvals under emergency labels (if a change is marked with an “emergency” label, the pipeline bypasses approvals to speed incident response)
- Shared tooling means a bad pipeline can impact multiple divisions - one team’s automation mistake can hurt other teams, too.

Observability Stack (Fed by the Datastore)

Centralized logging, metrics, tracing, plus incident alerting:

- Application Performance Monitoring dashboards
- SLO/SLA tracking
- KRI/KPI extraction
- incident retrospectives

Tech Stack

- Prometheus/Grafana + ELK/Splunk + OpenTelemetry
- pager/incident tool (PagerDuty/OpsGenie)
- centralized alert rules + per-product overlays
- long-term metrics stored in NoSQL/S3

Real-world posture

- Some telemetry pipelines store intermediate state in NoSQL.
- When NoSQL goes unhealthy, dashboards become stale or disappear.
- Teams argue about which alerts are “platform vs product.”