Compute & Application Layer

Droplet/container inventory, sizing, per-Compartment isolation mapping to infrastructure resources

Chapter 6 — Compute & Application Layer

6.1 Compute Model

The Platform runs on a small fixed droplet fleet with **containerized workloads under Docker Compose** — no Kubernetes, no managed PaaS. Rationale: the fleet is ≤5 hosts with a single-application production profile; Compose delivers the required per-Compartment network isolation (§6.4) with a fraction of the operational surface of an orchestrator, and its state is fully expressible in versioned files (IaC, Chapter 10). Revisit only if the Compartment count or droplet fleet grows materially (Chapter 16 thresholds).

As-built baseline: Docker 29.1.3 + Compose v2.40.3 installed from Ubuntu official repos (no third-party apt sources — consistent with the supply-chain posture, Chapter 11) on `sbsdash-server-prod`, `Dev-SBS-server`, and `monitor-servers` (2026-07-18); `docker` service enabled at boot. The wiki droplet intentionally has no Docker — BookStack runs as a system service and stays that way until the NYC2 rebuild, where containerization is decided (§6.6).

6.2 Droplet Roles and Sizing

Droplet	Size (as built)	Workload	Sizing assessment
`sbsdash-server-prod`	8 GB / 4 vCPU / 160 GB	nginx edge (Ch. 5), Compartment application containers, data services (interim, §6.5)	Adequate for build phase and first ~3 Compartments. Memory is the binding constraint once PostgreSQL + MongoDB + Redis colocate; Chapter 16 sets the resize/split trigger.
`Dev-SBS-server`	8 GB / 4 vCPU / 160 GB	Mirror of prod stack, synthetic data	Deliberate 1:1 with prod so Dev verification is load-meaningful.
`monitor-servers`	4 GB / 2 vCPU / 120 GB	Prometheus + Grafana + nginx; bastion (§3.3)	Adequate; Prometheus retention sizing in Ch. 12. No application workloads permitted — bastion role keeps this host minimal.
`sbs-wiki`	2 GB / 1 vCPU / 60 GB	BookStack	Adequate; carries at NYC2 rebuild.

Rule: the monitor and wiki are **single-purpose hosts**. Application or data workloads never deploy to them — the monitor because it is the bastion and alerting authority (compromise radius, §3.3),

the wiki because it is an operational surface with no tenant data (§1.2).

6.3 Runtime Toolchain

Target toolchain on prod and dev (Build Register items; **not yet installed** as of 2026-07-18):

Component	Purpose	Install source	Pinning
Node.js (LTS) + pnpm	Platform application runtime and package manager	NodeSource or nvm per Ch. 11 decision	Exact version in `.nvmrc` / `engines`; pnpm via corepack, version pinned
Rust + Cargo	Performance-critical platform components	rustup, pinned toolchain file (`rust-toolchain.toml`)	Exact toolchain per repo
Python 3	Tooling, ETL, operational scripts	Ubuntu system python + venvs	Per-project `requirements.txt` hash-pinned
Git	Deploy pulls, IaC	Ubuntu repo	—

Two rules carried from §2.4: versions are **identical on prod and dev** (drift = build defect), and *runtimes exist on the host primarily for build/operational tooling* — application code executes **inside containers**, whose images pin their own runtime versions (Ch. 11). Host-level Node/Rust/Python are not a second execution path for platform services.

6.4 Per-Compartment Isolation at the Compute Layer

This section realizes the application/network/data slices of the eight-layer isolation model (§1.2) in Compose terms. Each of the ten Compartments deploys as an isolated stack:

- **One Compose project per Compartment** (`compartment-01-hr` … `compartment-10-pr`), each with its own compose file rendered from a common IaC template — no shared "mega-compose."
- **One dedicated bridge network per Compartment** (`net-c01` … `net-c10`), `internal: true` where the Compartment needs no direct egress (egress instead via its declared integration endpoints per §3.4 chains). No container joins two Compartment networks. Cross-Compartment traffic is denied by non-membership — there is no route to filter because there is no shared segment.
- **Edge attachment:** each Compartment's frontend container additionally joins a shared `net-edge` network reachable **only** by nginx. nginx is the sole multi-network process; Compartment-to-Compartment via the edge network is impossible because only frontends and nginx sit on it and frontends accept only proxied requests (verified by header check at the app layer, APP-05).
- **Per-Compartment resource limits:** `mem_limit` and `cpus` set per service so one Compartment's runaway agent load cannot starve the other nine (availability isolation, not just confidentiality).
- **No privileged containers, no docker.sock mounts, `no-new-privileges: true`, read-only root filesystems where the app allows** — container escape hardening baseline.

- **Secrets:** injected at runtime from Vault (Ch. 8) via env-file rendered at deploy, never baked into images, never in compose files committed to the repo (INF-02). Per-Compartment Vault namespaces mean Compartment 03's containers can never read Compartment 08's credentials (AI-03 credential isolation at the infra layer).
- **AnythingMCP** deploys as a per-Compartment sidecar container on that Compartment's network — MCP tool servers are Compartment-scoped, which is the infrastructure half of AI-02 (tool allowlisting): a Compartment can only reach the MCP tools deployed on its own network, and each MCP container's outbound is bound by that Compartment's §3.4 egress chain.

6.5 Data Services Placement (Interim)

Pending the NYC2 Managed Database availability check (§2.3), PostgreSQL, MongoDB, and Redis run as containers on `sbsdash-server-prod`:

- Each data service on its own compose network (`net-data-pg`, etc.); Compartment application containers join **only** the data networks they are entitled to, and per-Compartment isolation inside each database engine (schemas/databases/ACLs per Compartment) is specified in Chapter 7.
- Data services bind container ports to the VPC-private interface or localhost only — never 0.0.0.0 (public-listener gate, §3.7).
- Volumes on the droplet's block storage with the backup regime of Chapter 14.
- If Managed Databases are confirmed in NYC2, migration moves data off-droplet and this section is amended by Change Order; the entitlement model (which Compartment reaches which database) is unchanged either way.

6.6 Wiki and Monitor Stacks

- **monitor-servers:** Prometheus/Grafana remain apt-installed system services (as built, verified) — containerizing the monitoring stack adds a Docker dependency to the thing that watches Docker; current form is deliberate. Docker exists on this host solely for operational tooling (e.g., blackbox exporter if containerized later).
- **sbs-wiki:** at NYC2 rebuild, BookStack is redeployed either as system service (snapshot restore, fastest) or containerized (compose template, consistent with fleet). Decision at rebuild; either way it stays a single-purpose host on the Prod VPC with private-bound exporter.

6.7 Deployment Flow

Images are built in CI from the GitHub repository (Ch. 10/11), pushed to the registry enumerated in the supply-chain allowlist, and pulled by prod over the §3.4 egress path. Deploys are `docker compose up -d` against IaC-rendered files, executed by the deploy user (per-person or CI identity — INF-04 remediation applies, §2.5); manual `docker run` on prod is prohibited and surfaces as drift (running-container set vs. compose state is a monitored comparison, MON-01). Rollback = redeploy previous pinned image tag; images are immutable and tagged by digest.

6.8 Verification

Gates (LH-SBS-INST-001): Compose project per Compartment present with dedicated network; cross-Compartment connectivity probe fails between any two Compartment networks; frontend reachable only via nginx (direct container port probe fails); no privileged container, no docker.sock mount, `no-new-privileges` set (docker inspect sweep); data-service ports absent from public interface; resource limits present on every Compartment service; running containers match compose state (drift check); toolchain versions identical prod↔dev (`node -v`, `rustc -V`, `python3 -V` diff).