THE ASCENT

One System, Every Architecture:
A Journey from Laptop to Planet-Scale

TECHNOVIZE
PUBLISHING

djangozen.com

The Ascent

One System, Every Architecture: A Journey from Laptop to Planet-Scale

KC Ramo

Technovize Publishing

The Ascent

One System, Every Architecture: A Journey from Laptop to Planet-Scale

Copyright © 2026 KC Ramo / Technovize Publishing. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise — without the prior written permission of the publisher, except for brief quotations in reviews. The code samples in this book may be used freely in your own projects.

First Edition, 2026

Published by Technovize Publishing
Amsterdam, The Netherlands

Cover and interior design by Technovize Publishing.

The information in this book is provided "as is", without warranty of any kind. While every effort has been made to ensure its accuracy, the author and publisher assume no responsibility for errors or omissions, or for damages resulting from the use of the information herein. Product and company names mentioned may be the trademarks of their respective owners.

Contents

Part III: Real Time and Real Big

About This Book

> *To every engineer who inherited a monolith and wondered where to begin.*

This book follows **Beacon**, a fictional collaborative knowledge platform, from a single Django process on a developer's laptop all the way to a planetary-scale distributed system serving millions of users across six continents. Every chapter introduces one scaling crisis, explains why it happened, and shows how to solve it — with real code, real frameworks, and real trade-offs.

Unlike survey-style system design books that jump between ten different products, *The Ascent* tells a single continuous story. The code evolves. The architecture deepens. The patterns compound. By the end, you will have built the intuition that senior engineers rely on when facing a system they have never seen before.

Who Should Read This Book

- **Software engineers with 1–5 years of experience** who understand basic web development and want to learn how systems grow beyond a single server.
- **Senior engineers and architects** who want a structured mental model for scaling decisions, with trade-offs made explicit rather than hand-waved.
- **Engineering managers and technical founders** who need to understand the architectural choices their teams are making — or avoiding.
- **Interview candidates** preparing for system design interviews, who want deeper understanding than a flashcard collection.

What You Need to Know

This book assumes you can read Python and have built at least one web application, even if it never left `localhost`. You do not need prior experience with Django, distributed systems, or any of the specific technologies introduced — each one is explained as it arrives.

How to Use This Book

Read it in order. Each chapter depends on the Beacon codebase as it stood at the end of the previous chapter. The complete source code for every chapter is available at the companion repository.

PART I

The Monolith

Chapters 1–4

CHAPTER 1

A Single Server and a Dream

> *Every system that serves a billion requests began as a single process running on a laptop nobody remembered the password to. This is the story of Beacon — and the story of every system you will ever build.*

1.1 The Birth of Beacon

It started, as these things always do, with a frustration.

Maya Chen was the third engineer at a forty-person startup that built data pipelines for climate researchers. The team had knowledge scattered across Slack threads, Notion pages, pinned Google Docs, and a Confluence instance that everyone hated but nobody had permission to delete. When a new hire asked where the production database credentials lived, five different people gave five different answers, and two of those answers were wrong.

Maya sketched an idea on a whiteboard: a single place where a team could write, organize, search, and collaborate on everything they knew. She called it **Beacon**.

The core insight was simple. Most knowledge tools force you to choose between *structure* (databases, schemas, rigid hierarchies) and *freedom* (documents, whiteboards, freeform notes). Beacon would refuse to make that choice. Every page would be both a human-readable document *and* a structured, queryable node in a knowledge graph. Tags, backlinks, and relationships would be first-class citizens, not afterthoughts. Real-time collaboration would work the way Google Docs worked. Search would understand context, not just keywords.

The vision was ambitious. But Maya had been an engineer long enough to know one thing: the right way to build an ambitious system is to start with the smallest thing that could possibly work.

She opened a terminal.

```
$ python -m venv beacon_env
$ source beacon_env/bin/activate
$ pip install django
$ django-admin startproject beacon .
$ python manage.py startapp core
```

A single directory. A handful of files. A development server that could handle exactly one request at a time. Beacon was born.

1.2 What We're Building

Before we write another line of code, let's be precise about what Beacon needs to do in its first incarnation. We are not building the planetary-scale system from Chapter 16. We are building something that Maya can show to her three teammates on Monday morning.

Beacon v0.1 — the minimum viable product:

Capability	What it means
Pages	Users can create, read, update, and delete documents. Each page has a title, body content, and an author.
Links	Pages can reference other pages. These references are stored as structured relationships, not just hyperlinks buried in Markdown.
Search	Basic full-text search across all page titles and bodies. Not Elasticsearch-grade — a SQL `LIKE` query is fine for now.
Users	Simple authentication. A user logs in, creates pages, and sees who wrote what.
Speed	Every page load completes in under 200 milliseconds. At ten concurrent users, nothing times out.

That is the entire specification. Notice what is *not* here: no real-time collaboration, no comments, no version history, no notifications, no rich text editor, no API, no mobile app, no analytics, no multi-tenancy, no horizontal scaling. All of that comes later, and each piece will arrive precisely when the system demands it.

This discipline — building only what the current scale requires — is the most important habit in system design. Premature architecture is technical debt with better marketing.

1.3 The Simplest Thing That Works

Let's set up Beacon. If you have worked with Django before, this will feel comfortingly familiar. If you have not, do not worry — I will explain every moving part as we encounter it.

Project structure

```
beacon/
├── manage.py
├── beacon/              # Project package (settings, URLs, WSGI)
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   └── asgi.py
├── core/                # Our application
│   ├── __init__.py
│   ├── models.py
│   ├── views.py
│   ├── urls.py
│   ├── admin.py
│   ├── templates/
│   │   └── core/
│   │       ├── base.html
│   │       ├── page_list.html
│   │       ├── page_detail.html
│   │       └── page_form.html
│   └── migrations/
│       └── __init__.py
└── db.sqlite3           # SQLite database (auto-created)
```

This is the canonical Django layout. The `beacon/` package holds project-wide configuration. The `core/` application holds our domain logic. In a larger system, we would have many applications — `search/`, `collab/`, `notifications/`, `analytics/` — but for v0.1, a single `core` app is all the separation we need. Do not split until the pain of the monolith exceeds the pain of the split.

Settings that matter

Django's `settings.py` is famously long, but only a handful of values matter for Beacon at this stage:

```
# beacon/settings.py (excerpt)

DEBUG = True  # Development only. In production, this is the first thing you turn
off.

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "core",                      # Our application
]

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

```
# For now, we use Django's default session and authentication backends.
# No Redis, no cache, no external dependencies beyond Django itself.
```

SQLite is the right database for v0.1. It requires zero configuration, stores everything in a single file, and can handle tens of thousands of reads per second on modest hardware. It is also, surprisingly, one of the most reliable pieces of software ever written — the SQLite test suite contains over 90 million lines of test code. The limitation, which we will confront directly in Chapter 2, is that SQLite handles concurrent writes poorly because it serializes them through a single write lock.

But today, we have ten users and maybe a hundred pages. SQLite will not break a sweat.

1.4 Models: The Shape of Knowledge

Django models are Python classes that map to database tables. They are the single most important design decision in any Django application because they encode your domain's structure at the database level, and the database is the hardest layer to change later.

Here is Beacon's initial model layer:

```
# core/models.py

from django.contrib.auth.models import User
from django.db import models
from django.urls import reverse


class Page(models.Model):
    """
    A single page in the Beacon knowledge graph.

    Each page has a title, a body (Markdown), an author, and timestamps.
    Pages can reference other pages through the PageLink model, forming
    the edges of our knowledge graph.
    """

    title = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255, unique=True, blank=True)
    body = models.TextField(blank=True, default="")
    author = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="pages",
    )
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ["-updated_at"]
        indexes = [
            models.Index(fields=["slug"]),
            models.Index(fields=["author", "-created_at"]),
            # Full-text search index placeholder.
```

```
            # SQLite's FTS5 extension will replace this in a future
            # migration. For now, LIKE queries suffice.
        ]

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse("core:page_detail", kwargs={"slug": self.slug})

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super().save(*args, **kwargs)


class PageLink(models.Model):
    """
    A directed edge in the Beacon knowledge graph.

    When page A references page B (e.g., via a [[wikilink]] in the body),
    we create a PageLink from A (source) to B (target). This lets us query
    relationships without parsing Markdown at read time.
    """

    source = models.ForeignKey(
        Page,
        on_delete=models.CASCADE,
        related_name="outgoing_links",
    )
    target = models.ForeignKey(
        Page,
        on_delete=models.CASCADE,
        related_name="incoming_links",
    )
    context = models.TextField(
        blank=True,
        default="",
        help_text="The surrounding text where the link appeared.",
    )
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ["source", "target"]
        indexes = [
            models.Index(fields=["source"]),
            models.Index(fields=["target"]),
        ]

    def __str__(self):
        return f"{self.source.title} → {self.target.title}"
```

Design decisions worth noticing

The `slug` field. Every page gets a URL-safe identifier derived from its title. A page titled "Deployment Runbook" becomes `/pages/deployment-runbook`. Slugs are unique, human-readable, and stable — even if the title changes later, the slug can stay the same. This is a small decision that pays compounding dividends: readable URLs make debugging easier, make logs searchable, and make APIs intuitive.

The `PageLink` model. This is where Beacon differentiates itself from a simple wiki. Instead of burying page relationships inside Markdown text, we extract them into a structured table. When a user writes `For details, see [[Deployment Runbook]]` in a page body, our save logic parses the wikilink syntax and creates a `PageLink` row. At query time, we can ask questions like "what pages link to this one?" or "what are the most-linked pages on the site?" with a simple database join instead of a full-text scan of every page body. This is an example of **write-time computation** — we do the parsing work once, at save time, so that reads stay fast.

Database indexes. We create indexes on `slug` (for URL lookups), `author + created_at` (for "pages by this user, newest first"), `source` (for outgoing link queries), and `target` (for backlink queries). Indexes are the cheapest performance investment you can make, but every index also slows down writes because the database must update the index structure on every insert and update. At our current scale, the write penalty is invisible. By Chapter 4, we will learn to be more selective.

1.5 Views: The Code That Answers Requests

A Django view is a Python function (or class) that receives an HTTP request and returns an HTTP response. It is the bridge between the outside world and your database. Beacon's initial views are deliberately simple — list, detail, create, update, delete. The CRUD pentagon.

```
# core/views.py

from django.contrib.auth.decorators import login_required
from django.contrib.auth.mixins import LoginRequiredMixin
from django.db.models import Count, Q
from django.shortcuts import get_object_or_404, redirect, render
from django.urls import reverse_lazy
from django.views.generic import (
    ListView,
    DetailView,
    CreateView,
    UpdateView,
    DeleteView,
)

from .models import Page, PageLink


class PageListView(LoginRequiredMixin, ListView):
    """
    The homepage of Beacon.

    Shows all pages, newest first, with the count of incoming links
    for each page — a lightweight "importance" signal.
    """

    model = Page
    template_name = "core/page_list.html"
```

```
    context_object_name = "pages"
    paginate_by = 50

    def get_queryset(self):
        return (
            Page.objects.annotate(link_count=Count("incoming_links"))
            .select_related("author")
            .order_by("-updated_at")
        )


class PageDetailView(LoginRequiredMixin, DetailView):
    """
    A single page, with its incoming and outgoing links rendered
    as structured navigation rather than inline hyperlinks.
    """

    model = Page
    template_name = "core/page_detail.html"
    context_object_name = "page"

    def get_object(self, queryset=None):
        return get_object_or_404(
            Page.objects.select_related("author").prefetch_related(
                "outgoing_links__target",
                "incoming_links__source",
            ),
            slug=self.kwargs["slug"],
        )


class PageCreateView(LoginRequiredMixin, CreateView):
    """Create a new page. The author is set to the current user."""

    model = Page
    template_name = "core/page_form.html"
    fields = ["title", "body"]

    def form_valid(self, form):
        form.instance.author = self.request.user
        response = super().form_valid(form)
        self._refresh_links(form.instance)
        return response

    def _refresh_links(self, page):
        """
        Parse the page body for [[wikilinks]] and create or remove
        PageLink rows to match. This keeps the knowledge graph in sync
        with the page content.
        """
        import re

        PageLink.objects.filter(source=page).delete()
        pattern = re.compile(r"\[\[([^\]]+)\]\]")
        matches = pattern.findall(page.body)

        for match in matches:
            # Try to find a page whose title or slug matches the link text.
            target = Page.objects.filter(
                Q(title__iexact=match) | Q(slug__iexact=match)
            ).first()
            if target and target != page:
                PageLink.objects.get_or_create(
                    source=page,
                    target=target,
```

```
                    defaults={"context": match},
                )


class PageUpdateView(LoginRequiredMixin, UpdateView):
    """Edit an existing page. Links are refreshed after save."""

    model = Page
    template_name = "core/page_form.html"
    fields = ["title", "body"]

    def form_valid(self, form):
        response = super().form_valid(form)
        # Reuse the link-refresh logic from PageCreateView.
        PageCreateView._refresh_links(self, form.instance)
        return response


class PageDeleteView(LoginRequiredMixin, DeleteView):
    """Delete a page and all its associated links."""

    model = Page
    template_name = "core/page_confirm_delete.html"
    success_url = reverse_lazy("core:page_list")


class PageSearchView(LoginRequiredMixin, ListView):
    """
    Full-text search across page titles and bodies.

    Uses SQL LIKE for now. The percentage signs (%) are SQL wildcards.
    A query for "deploy" becomes: WHERE title LIKE '%deploy%' OR body LIKE
'%deploy%'

    This is fast enough for thousands of pages. For tens of thousands,
    we will need PostgreSQL full-text search or Elasticsearch (Chapter 10).
    """

    model = Page
    template_name = "core/page_list.html"
    context_object_name = "pages"

    def get_queryset(self):
        query = self.request.GET.get("q", "").strip()
        if not query:
            return Page.objects.none()

        return (
            Page.objects.filter(
                Q(title__icontains=query) | Q(body__icontains=query)
            )
            .select_related("author")
            .order_by("-updated_at")
        )

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["search_query"] = self.request.GET.get("q", "")
        return context
```

Why class-based views?

Django offers two ways to write views: functions and classes. Beacon uses class-based views (CBVs) for a reason that will matter enormously by Chapter 7. A CBV is a collection of small, overridable methods — `get_queryset`, `get_context_data`, `form_valid` — that each do one thing. When we later need to add caching, rate limiting, or audit logging to every page view, we can inject that behavior into a base class once rather than modifying fifty individual functions. This is the **template method pattern**, and it is the same design principle that makes middleware, decorators, and service meshes work at higher scales.

For now, the important thing is that each view does exactly one conceptual job and no more. `PageDetailView` fetches a page by slug. `PageSearchView` filters pages by a query string. The link-refresh logic in `PageCreateView` is the only complex piece, and we will extract it into a proper service layer once it grows beyond a single method.

1.6 URL Routing: Mapping Requests to Views

Django's URL configuration is a dispatch table — a mapping from URL patterns to view callables. For Beacon v0.1, it is a single file:

```
# core/urls.py

from django.urls import path

from . import views

app_name = "core"

urlpatterns = [
    path("", views.PageListView.as_view(), name="page_list"),
    path("search/", views.PageSearchView.as_view(), name="page_search"),
    path("pages/new/", views.PageCreateView.as_view(), name="page_create"),
    path("pages/<slug:slug>/", views.PageDetailView.as_view(), name="page_detail"),
    path("pages/<slug:slug>/edit/", views.PageUpdateView.as_view(),
name="page_update"),
    path("pages/<slug:slug>/delete/", views.PageDeleteView.as_view(),
name="page_delete"),
]
```

And the project-level router includes it:

```
# beacon/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
```

```
    path("accounts/", include("django.contrib.auth.urls")),
    path("", include("core.urls")),
]
```

The URL structure is deliberately shallow. `/pages/deployment-runbook` rather than `/spaces/engineering/projects/infra/pages/deployment-runbook`. Deep hierarchies are hard to reorganize and force you to know a page's location before you can navigate to it. Flat namespaces — with slugs as unique identifiers — are simpler, more flexible, and easier to shard later. We will add organizational structures (spaces, teams, tags) in Chapter 3, but they will be query-time groupings layered on top of flat slugs, not path-based hierarchies.

1.7 Templates: The HTML Nobody Sees

Django's template language is deliberately underpowered. You cannot call arbitrary Python functions from a template. You cannot import modules. You get variables, filters, loops, conditionals, and template inheritance. This constraint is a feature: it forces business logic into views and models where it can be tested, leaving templates to do nothing but render data.

Here is the base template that every Beacon page extends:

```
<!-- core/templates/core/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Beacon{% endblock %}</title>
    <style>
        /* Inline styles for v0.1. We'll add a proper asset pipeline later. */
        :root {
            --bg: #fafafa;
            --surface: #ffffff;
            --text: #1a1a2e;
            --muted: #6b7280;
            --accent: #2563eb;
            --border: #e5e7eb;
            --radius: 8px;
            --shadow: 0 1px 3px rgba(0, 0, 0, 0.08);
        }
        *, *::before, *::after { box-sizing: border-box; margin: 0; padding: 0; }
        body {
            font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,
sans-serif;
            background: var(--bg);
            color: var(--text);
            line-height: 1.6;
            max-width: 800px;
            margin: 0 auto;
            padding: 2rem 1rem;
        }
        header {
            display: flex;
            justify-content: space-between;
            align-items: center;
```

```
            margin-bottom: 2rem;
            padding-bottom: 1rem;
            border-bottom: 1px solid var(--border);
        }
        header h1 { font-size: 1.5rem; font-weight: 700; }
        header nav { display: flex; gap: 1rem; align-items: center; }
        a { color: var(--accent); text-decoration: none; }
        a:hover { text-decoration: underline; }
        .page-card {
            background: var(--surface);
            border: 1px solid var(--border);
            border-radius: var(--radius);
            padding: 1.25rem;
            margin-bottom: 1rem;
            box-shadow: var(--shadow);
        }
        .page-card h2 { font-size: 1.125rem; margin-bottom: 0.25rem; }
        .page-card .meta { font-size: 0.8rem; color: var(--muted); }
        .page-body {
            background: var(--surface);
            border: 1px solid var(--border);
            border-radius: var(--radius);
            padding: 2rem;
            margin-bottom: 1.5rem;
            box-shadow: var(--shadow);
            line-height: 1.8;
        }
        .links-sidebar {
            background: var(--surface);
            border: 1px solid var(--border);
            border-radius: var(--radius);
            padding: 1.25rem;
            margin-bottom: 1.5rem;
            box-shadow: var(--shadow);
        }
        .links-sidebar h3 { font-size: 0.9rem; color: var(--muted); margin-bottom:
0.5rem; }
        .links-sidebar ul { list-style: none; }
        .links-sidebar li { padding: 0.25rem 0; }
        form { display: flex; flex-direction: column; gap: 1rem; }
        input, textarea {
            width: 100%;
            padding: 0.75rem;
            border: 1px solid var(--border);
            border-radius: var(--radius);
            font-family: inherit;
            font-size: 1rem;
        }
        textarea { min-height: 300px; resize: vertical; }
        button, .btn {
            display: inline-block;
            padding: 0.6rem 1.25rem;
            background: var(--accent);
            color: white;
            border: none;
            border-radius: var(--radius);
            font-size: 0.95rem;
            cursor: pointer;
            text-decoration: none;
        }
        button:hover, .btn:hover { opacity: 0.9; text-decoration: none; }
        .btn-secondary { background: var(--border); color: var(--text); }
        .search-form { display: flex; gap: 0.5rem; flex-direction: row; }
        .search-form input { flex: 1; }
        .empty-state {
```

```
            text-align: center;
            padding: 4rem 1rem;
            color: var(--muted);
        }
        .pagination { display: flex; gap: 0.5rem; justify-content: center; margin-
top: 1.5rem; }
        .messages { margin-bottom: 1rem; }
        .message {
            padding: 0.75rem 1rem;
            border-radius: var(--radius);
            margin-bottom: 0.5rem;
            background: #dbeafe;
            color: #1e40af;
        }
    </style>
</head>
<body>
    <header>
        <h1><a href="{% url 'core:page_list' %}">   Beacon</a></h1>
        <nav>
            <form class="search-form" method="get" action="{% url
'core:page_search' %}">
                <input type="text" name="q" placeholder="Search pages…"
                       value="{{ search_query|default:'' }}">
                <button type="submit">Search</button>
            </form>
            <a href="{% url 'core:page_create' %}" class="btn">+ New Page</a>
            {% if user.is_authenticated %}
                <span>{{ user.username }}</span>
                <form method="post" action="{% url 'logout' %}"
style="display:inline">
                    {% csrf_token %}
                    <button type="submit" class="btn-secondary">Logout</button>
                </form>
            {% endif %}
        </nav>
    </header>

    {% if messages %}
    <div class="messages">
        {% for message in messages %}
            <div class="message">{{ message }}</div>
        {% endfor %}
    </div>
    {% endif %}

    <main>
        {% block content %}{% endblock %}
    </main>
</body>
</html>
```

And the page detail template that renders a single page with its knowledge graph context:

```
<!-- core/templates/core/page_detail.html -->
{% extends "core/base.html" %}

{% block title %}{{ page.title }} — Beacon{% endblock %}

{% block content %}
<article class="page-body">
    <h1>{{ page.title }}</h1>
    <p class="meta">
        By {{ page.author.username }} ·
```

```
        Updated {{ page.updated_at|date:"M j, Y" }} ·
        <a href="{% url 'core:page_update' page.slug %}">Edit</a> ·
        <a href="{% url 'core:page_delete' page.slug %}">Delete</a>
    </p>
    <hr style="margin: 1rem 0; border-color: var(--border);">
    <div>
        {{ page.body|linebreaks }}
    </div>
</article>

{% if page.outgoing_links.exists or page.incoming_links.exists %}
<div class="links-sidebar">
    {% if page.outgoing_links.exists %}
    <h3>  Links to</h3>
    <ul>
        {% for link in page.outgoing_links.all %}
        <li>
            <a href="{% url 'core:page_detail' link.target.slug %}">
                {{ link.target.title }}
            </a>
            {% if link.context %}
                <span style="color: var(--muted); font-size: 0.85rem;">
                    — "{{ link.context }}"
                </span>
            {% endif %}
        </li>
        {% endfor %}
    </ul>
    {% endif %}

    {% if page.incoming_links.exists %}
    <h3>  Linked from</h3>
    <ul>
        {% for link in page.incoming_links.all %}
        <li>
            <a href="{% url 'core:page_detail' link.source.slug %}">
                {{ link.source.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
    {% endif %}
</div>
{% endif %}

<p><a href="{% url 'core:page_list' %}">← Back to all pages</a></p>
{% endblock %}
```

The page list template shows every page as a card, ordered by recency:

```
<!-- core/templates/core/page_list.html -->
{% extends "core/base.html" %}

{% block title %}
    {% if search_query %}Search: {{ search_query }}{% else %}Beacon{% endif %}
{% endblock %}

{% block content %}
{% if search_query %}
    <h2>Results for "{{ search_query }}"</h2>
{% endif %}

{% if pages %}
    {% for page in pages %}
```

```
    <div class="page-card">
        <h2><a href="{% url 'core:page_detail' page.slug %}">{{ page.title }}</a></
h2>
        <p class="meta">
            By {{ page.author.username }} ·
            {{ page.updated_at|date:"M j, Y" }} ·
            {% if page.link_count %}{{ page.link_count }} link{{ page.link_count|
pluralize }}{% endif %}
        </p>
    </div>
    {% endfor %}

    {% if is_paginated %}
    <div class="pagination">
        {% if page_obj.has_previous %}
            <a href="?page={{ page_obj.previous_page_number }}" class="btn-
secondary">← Newer</a>
        {% endif %}
        <span style="padding: 0.6rem 1rem;">Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}</span>
        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}" class="btn-
secondary">Older →</a>
        {% endif %}
    </div>
    {% endif %}
{% else %}
    <div class="empty-state">
        <h2>No pages yet</h2>
        <p>Create the first page to start building your team's knowledge graph.</p>
        <p><a href="{% url 'core:page_create' %}" class="btn">+ New Page</a></p>
    </div>
{% endif %}
{% endblock %}
```

1.8 The Request Journey

Let us follow a single HTTP request through Beacon from start to finish. Understanding this flow is the foundation of every performance optimization, every debugging session, and every architectural decision in the chapters ahead.

A user types `https://beacon.example.com/pages/deployment-runbook` into their browser and presses Enter. Here is what happens:

Step 1: DNS resolution

The browser asks the operating system: "what IP address corresponds to `beacon.example.com`?" The OS checks its local cache, then queries a recursive DNS resolver (usually the ISP's, or a public one like 8.8.8.8). The resolver walks the DNS hierarchy — root servers, `.com` TLD servers, `example.com` authoritative nameservers — and eventually returns an IP address like `198.51.100.42`. This entire dance typically takes **20 to 120 milliseconds** on a cold lookup and **under 5 milliseconds** when the result is cached.

At our current scale, DNS is a non-issue. By Chapter 13, when Beacon spans five continents, DNS will become a critical part of the latency budget and a potential failure domain.

Step 2: TCP handshake

The browser opens a TCP connection to `198.51.100.42:443` (HTTPS uses port 443 by default). The three-way handshake — SYN, SYN-ACK, ACK — takes one round-trip time (RTT). If the server is in the same region, that is **5 to 30 milliseconds**. If the server is on another continent, it can be **150 to 300 milliseconds** before a single byte of application data has been exchanged.

Step 3: TLS handshake

HTTPS requires a Transport Layer Security handshake on top of TCP. TLS 1.3 — the current standard — completes in one additional RTT (two more messages) for a total of **two RTTs** before the first HTTP request can be sent. TLS 1.2 took two RTTs (three total including TCP), which is one reason the industry moved aggressively to TLS 1.3.

Step 4: HTTP request

The browser sends an HTTP request:

```
GET /pages/deployment-runbook HTTP/1.1
Host: beacon.example.com
Accept: text/html
Cookie: sessionid=a8f3c9e1b2d4...
```

This is roughly **500 to 1500 bytes** of headers. The transmission time is negligible at this size — under 1 millisecond on any modern connection.

Step 5: WSGI gateway

The HTTP request arrives at a WSGI server — in development, Django's built-in `runserver`; in production, something like **Gunicorn** or **uWSGI**. The WSGI server is a protocol adapter: it translates raw HTTP into a Python dictionary (`environ`) and passes it to Django's WSGI handler. This translation takes **a few microseconds**.

Step 6: Django middleware stack

Django processes the request through its middleware pipeline. Each middleware class is a wrapper around the next one, forming an onion:

```
Request  →  SecurityMiddleware
         →  SessionMiddleware
         →  CommonMiddleware
         →  CsrfViewMiddleware
         →  AuthenticationMiddleware
```

```
          →  MessageMiddleware
          →  Our View
Response ←  (back through each layer in reverse)
```

Each middleware can inspect, modify, or short-circuit the request. `SessionMiddleware` reads the `sessionid` cookie, queries the session store (SQLite, for now), and attaches a `request.user` object. `AuthenticationMiddleware` confirms the user exists and is active. If any middleware raises an exception or returns a response early, the remaining layers are skipped.

The middleware stack is one of Django's most powerful abstractions, and it will become the insertion point for caching (Chapter 3), rate limiting (Chapter 4), distributed tracing (Chapter 14), and feature flags (Chapter 15).

Step 7: URL routing and view dispatch

Django's URL resolver matches `/pages/deployment-runbook` against the patterns in `core/urls.py`. It finds:

```
path("pages/<slug:slug>/", views.PageDetailView.as_view(), name="page_detail")
```

The `<slug:slug>` converter extracts `deployment-runbook` from the path. Django instantiates `PageDetailView` and calls its `dispatch()` method, which routes to `get()` (for an HTTP GET request).

Step 8: Database query

`PageDetailView.get_object()` executes a SQL query:

```
SELECT "core_page"."id",
       "core_page"."title",
       "core_page"."slug",
       "core_page"."body",
       "core_page"."author_id",
       "core_page"."created_at",
       "core_page"."updated_at",
       "auth_user"."id",
       "auth_user"."username",
       "auth_user"."email"
  FROM "core_page"
  JOIN "auth_user"
    ON ("core_page"."author_id" = "auth_user"."id")
 WHERE "core_page"."slug" = 'deployment-runbook'
```

Django translates this into a Python `Page` object. The entire query takes **0.2 to 2 milliseconds** against SQLite for a database with a few thousand pages, assuming the `slug` index is used (and it is — we created it in the model's `Meta.indexes`).

Step 9: Template rendering

Django loads `core/templates/core/page_detail.html`, compiles it to an internal node tree (cached after the first render), and evaluates it with the context dictionary containing `page`, `user`, and any other variables passed by the view. Template rendering for a page of this size takes **1 to 5 milliseconds**.

Step 10: HTTP response

Django returns an `HttpResponse` object with the rendered HTML (typically **5 to 30 kilobytes**). The WSGI server serializes it to bytes and sends it over the TCP connection. The browser receives it, parses the HTML, discovers any additional resources (CSS, JavaScript, images), and begins fetching them in parallel.

Total server-side time for this request: 5 to 15 milliseconds. The user perceives the page as "loaded" in roughly **50 to 300 milliseconds**, depending primarily on network latency.

1.9 Where Does the Time Go?

Here is a latency budget for the deployment-runbook page load, assuming the user is in the same region as the server:

Component	Time (ms)	Percentage
DNS lookup (cached)	1	< 1%
TCP handshake	10	8%
TLS handshake	8	6%
HTTP request transmission	1	< 1%
Django middleware	2	2%
Database query	1	< 1%
Template rendering	3	2%
Response transmission	5	4%
Network round-trips (3 × RTT)	**90**	**72%**
Total	**~125**	**100%**

The server-side work is **6 milliseconds**. The network is **90 milliseconds**. This ratio — the network dominates the server — holds true for almost every web application at small scale. It is why the first performance optimization for any system is usually "put the server closer to the users" (a CDN, a closer region), not "make the code faster."

But the ratio will flip. By Chapter 4, when Beacon has a hundred thousand pages and a thousand concurrent users, the database query will take 50 milliseconds, the template will take 30 milliseconds, and the middleware stack will have acquired new layers for caching and rate limiting. The server-side time will grow from 6 to 200 milliseconds, and suddenly the code will matter as much as the network.

Understanding your current latency budget — and knowing which number will grow first — is the difference between optimizing the right thing and optimizing the thing that feels technical.

1.10 A Single Point of Failure

Beacon v0.1 runs on one server: a single EC2 instance, a single DigitalOcean droplet, or Maya's laptop during development. The server runs Django, Gunicorn, and SQLite — all on the same machine, all in the same process space.

This architecture has exactly one advantage: **simplicity**. There is one thing to deploy, one thing to monitor, one thing to debug. The entire system fits in Maya's head. When something breaks, the surface area of the problem is small.

It also has exactly one disadvantage: **everything is a single point of failure**.

If the database file becomes corrupted, all data is lost. If the server runs out of disk space, the entire application stops. If the network cable is unplugged, nobody can reach Beacon. If Maya deploys a bug that causes an infinite loop, the server's CPU pegs at 100%, and every user experiences the outage simultaneously.

The formal term for this architecture is a **single-tier deployment** — compute and storage are co-located on one machine. It is the correct architecture for:

- Prototypes and MVPs
- Internal tools with fewer than 50 users
- Applications where downtime is annoying but not catastrophic
- Teams with one or two engineers who need to move fast

It is the *incorrect* architecture for:

- Any system that stores data you cannot afford to lose
- Any system that must be available 24/7
- Any system with paying customers who expect an SLA
- Any system where the data is more valuable than the servers

Beacon is about to discover which category it belongs in.

1.11 The First Deployment

Maya deploys Beacon to a small virtual machine. The deployment is manual — she SSHs into the server, pulls the latest code from Git, runs migrations, and restarts Gunicorn. She writes a shell script called `deploy.sh` that does all of this in sequence:

```
#!/bin/bash
# deploy.sh — Beacon v0.1 deployment script
# Run from the project root on the production server.

set -euo pipefail

echo "=== Deploying Beacon ==="

cd /opt/beacon
git pull origin main

source /opt/beacon/beacon_env/bin/activate
pip install -r requirements.txt

python manage.py migrate --noinput
python manage.py collectstatic --noinput

# Gracefully restart Gunicorn by sending SIGHUP to the master process.
# This tells Gunicorn to start new workers and gracefully shut down old ones,
# achieving zero-downtime restarts for a single-server deployment.
kill -HUP $(cat /var/run/beacon_gunicorn.pid)

echo "=== Deploy complete ==="
```

This script is not elegant. It has no rollback mechanism, no health check before the restart, and no notification if something goes wrong. But for a single-server deployment with ten users, it is good enough. The cost of building a CI/CD pipeline right now exceeds the cost of the occasional five-minute outage during a deployment.

This trade-off — *the cost of the solution must not exceed the cost of the problem* — will recur in every chapter of this book. It is the single principle that separates pragmatic system design from resume-driven architecture.

1.12 The First Growth Signal

Two weeks after launch, Beacon has 47 users. The engineering team uses it for runbooks. The product team uses it for feature specs. Someone in marketing discovered it and added competitive research. The database has 312 pages and 1,847 links between them.

Maya notices something in the Django debug toolbar: the page list view — the homepage — now takes 85 milliseconds to render, up from 12 milliseconds at launch. The culprit is the `link_count` annotation in `PageListView.get_queryset()`:

```
Page.objects.annotate(link_count=Count("incoming_links"))
```

For every page in the list, Django performs a subquery to count incoming links. With 312 pages, that is not catastrophic — but it is a linear scan over a growing `PageLink` table. At 1,000 pages, the query will take 200 milliseconds. At 10,000 pages, it will take multiple seconds.

This is the first growth signal. The system is not broken. Nobody has complained about performance. But the trend is unmistakable: the current architecture will not survive the next order of magnitude.

Maya opens a new file called `notes/performance.md` and writes:

> ***Page list link_count — N+1 count subquery.*** *Options: 1. Add a `link_count` column to the Page model and update it on save (denormalization). 2. Cache the page list view (Redis or Django's cache framework). 3. Move to a materialized view refreshed periodically. 4. Do nothing until someone complains.*
>
> *Decision: wait. 85 ms is fine. Revisit at 500 pages or 200 ms, whichever comes first.*

This note — a performance observation with a threshold for action — is the simplest and most effective monitoring system in existence. It costs nothing to maintain, it captures the context that a CPU graph never will, and it prevents premature optimization while ensuring that real problems are not ignored.

1.13 What We Built

Beacon v0.1 is complete. Here is what exists:

Component	Technology	Purpose
Web framework	Django 5.x	HTTP handling, routing, ORM, auth
Database	SQLite	Primary data store
Deployment	Single VM, Gunicorn, manual deploy script	Serving requests
Monitoring	Django Debug Toolbar, Maya's notes	Performance awareness

And here is what we deliberately did *not* build:

- No caching layer (Chapter 3)
- No read replicas (Chapter 5)
- No background workers (Chapter 8)
- No WebSocket support (Chapter 9)
- No search engine (Chapter 10)
- No multi-region deployment (Chapter 13)

Each missing piece is an intentional decision, not an oversight. The system is as simple as it can be while still being useful. When load increases, we will know exactly which limit we hit first, because there is only one server and one database — the failure surface is small and the diagnosis is fast.

1.14 The Principles So Far

Before we move on, let us extract the principles this chapter has demonstrated:

1. Start with the smallest thing that works

A Django project with one app, SQLite, and manual deployment is the correct architecture for an internal tool with under 50 users. Do not build for scale you do not have.

2. Model your domain carefully

The `Page` and `PageLink` models encode Beacon's core insight — that knowledge is a graph, not a folder hierarchy — in two database tables. Get the data model right early, because changing it later is expensive. Changing code is cheap.

3. Understand your latency budget

Before optimizing anything, measure where time actually goes. In v0.1, the network accounts for 72% of page load latency. Optimizing the database query from 1 ms to 0.5 ms would be invisible to users. Understanding this saves you from optimizing the wrong thing.

4. Denormalize at write time

Parsing wikilinks on save — rather than scanning every page body on every read — is an example of write-time computation. Do the work once, when data is created, so that reads stay fast. This pattern will reappear in caching, materialized views, and event sourcing.

5. Set thresholds, not just metrics

Maya's note — "revisit at 500 pages or 200 ms" — is more valuable than a dashboard full of graphs. A metric without a threshold is noise. A threshold without a plan is anxiety. Both together are a monitoring system.

6. The cost of the solution must not exceed the cost of the problem

The manual deploy script is fine for now. The `LIKE`-based search is fine for now. Single-server, single-database, single point of failure — all of it is fine for now. "For now" is not an excuse; it is an engineering judgment. Knowing when "for now" ends is the skill.

1.15 What's Next

Chapter 2 follows Beacon as it crosses the threshold from an internal tool to a company-wide platform. The user count triples. The page count crosses 5,000. The `link_count` annotation — the one Maya decided not to optimize — finally breaks.

We will learn:

- How to profile a Django application under real load using Locust
- Why the N+1 query problem is the most common performance bug in ORM-based applications, and how `select_related` and `prefetch_related` solve it
- How PostgreSQL compares to SQLite when concurrency matters
- Why connection pooling is essential and how `pgbouncer` works
- The difference between *latency* and *throughput* — and why it is the most important distinction in system performance

End of sample

You have read the opening chapter of

The Ascent

The complete book continues from here — every chapter, every appendix, in PDF and EPUB.

EUR 44.00

Get the full book

https://djangozen.com/ebooks/book/the-ascent/