Extracted from:

Release It! Second Edition

Design and Deploy Production-Ready Software

This PDF file contains pages extracted from *Release It! Second Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication 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 consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Release It!

Second Edition

Design and Deploy Production-Ready Software

Michael T. Nygard

Edited by Katharine Dvorak

Release It! Second Edition

Design and Deploy Production-Ready Software

Michael T. Nygard

The Pragmatic Bookshelf

Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication 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 consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-239-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

CHAPTER 1

Living in Production

You've worked hard on your project. It looks like all the features are actually complete, and most even have tests. You can breathe a sigh of relief. You're done.

Or are you?

Does "feature complete" mean "production ready"? Is your system really ready to be deployed? Can it be run by operations and face the hordes of real-world users without you? Are you starting to get that sinking feeling that you'll be faced with late-night emergency phone calls and alerts? It turns out there's a lot more to development than just adding all the features.

Software design as taught today is terribly incomplete. It only talks about what systems *should* do. It doesn't address the converse—what systems should *not* do. They should not crash, hang, lose data, violate privacy, lose money, destroy your company, or kill your customers.

Too often, project teams aim to pass the quality assurance (QA) department's tests instead of aiming for life in production. That is, the bulk of your work probably focuses on passing testing. But testing—even agile, pragmatic, automated testing—is not enough to prove that software is ready for the real world. The stresses and strains of the real world, with crazy real users, globe-spanning traffic, and virus-writing mobs from countries you've never even heard of go well beyond what you could ever hope to test for.

But first, you will need to accept the fact that despite your best laid plans, bad things will still happen. It's always good to prevent them when possible, of course. But it can be downright fatal to assume that you've predicted and eliminated all possible bad events. Instead, you want to take action and prevent the ones you can but make sure that your system as a whole can recover from whatever unanticipated, severe traumas might befall it.

Aiming for the Right Target

Most software is designed for the development lab or the testers in the QA department. It is designed and built to pass tests such as, "The customer's first and last names are required, but the middle initial is optional." It aims to survive the artificial realm of QA, not the real world of production.

Software design today resembles automobile design in the early '90s—disconnected from the real world. Cars designed solely in the cool comfort of the lab looked great in models and CAD systems. Perfectly curved cars gleamed in front of giant fans, purring in laminar flow. The designers inhabiting these serene spaces produced designs that were elegant, sophisticated, clever, fragile, unsatisfying, and ultimately short-lived. Most software architecture and design happens in equally clean, distant environs.

Do you want a car that looks beautiful but spends more time in the shop than on the road? Of course not! You want to own a car designed for the real world. You want a car designed by somebody who knows that oil changes are *always* 3,000 miles late, that the tires must work just as well on the last sixteenth of an inch of tread as on the first, and that you will certainly, at some point, stomp on the brakes while holding an Egg McMuffin in one hand and a phone in the other.

When our system passes QA, can we say with confidence that it's ready for production? Simply passing QA tells us little about the system's suitability for the next three to ten years of life. It could be the Toyota Camry of software, racking up thousands of hours of continuous uptime. Or it could be the Chevy Vega (a car whose front end broke off on the company's own test track) or the Ford Pinto (a car prone to blowing up when hit in just the right way). It's impossible to tell from a few days or even a few weeks of testing what the next several years will bring.

Product designers in manufacturing have long pursued "design for manufacturability"—the engineering approach of designing products such that they can be manufactured at low cost and high quality. Prior to this era, product designers and fabricators lived in different worlds. Designs thrown over the wall to production included screws that could not be reached, parts that were easily confused, and custom parts where off-the-shelf components would serve. Inevitably, low quality and high manufacturing cost followed.

We're in a similar state today. We end up falling behind on the new system because we're constantly taking support calls from the last half-baked project we shoved out the door. Our analog of "design for manufacturability" is "design for production." We don't hand designs to fabricators, but we do hand finished software to IT operations. We need to design individual software systems, and the whole ecosystem of interdependent systems, to operate at low cost and high quality.

The Scope of the Challenge

In the easy, laid-back days of client/server systems, a system's user base would be measured in the tens or hundreds, with a few dozen concurrent users at most. Today we routinely see active user counts larger than the population of entire continents. And I'm not just talking about Antarctica and Australia here! We've seen our first billion-user social network, and it won't be the last.

Uptime demands have increased too. Whereas the famous "five nines" (99.999 percent) uptime was once the province of the mainframe and its caretakers, even garden-variety commerce sites are now expected to be available 24 by 7 by 365. (That phrase has always bothered me. As an engineer, I expect it to either be "24 by 365" or be "24 by 7 by 52.") Clearly, we've made tremendous strides even to consider the scale of software built today; but with the increased reach and scale of our systems come new ways to break, more hostile environments, and less tolerance for defects.

The increasing scope of this challenge—to build software fast that's cheap to build, good for users, and cheap to operate—demands continually improving architecture and design techniques. Designs appropriate for small WordPress websites fail outrageously when applied to large scale, transactional, distributed systems, and we'll look at some of those outrageous failures.

A Million Dollars Here, a Million Dollars There

A lot is on the line here: your project's success, your stock options or profit sharing, your company's survival, and even your job. Systems built for QA often require so much ongoing expense, in the form of operations cost, downtime, and software maintenance, that they never reach profitability, let alone net positive cash for the business (reached only after the profits generated by the system pay back the costs incurred in building it.) These systems exhibit low availability, direct losses in missed revenue, and indirect losses through damage to the brand.

During the hectic rush of a development project, you can easily make decisions that optimize development cost at the expense of operational cost. This makes sense only in the context of the team aiming for a fixed budget and delivery date. In the context of the organization paying for the software, it's a bad choice. Systems spend much more of their life in operation than in development—at least, the ones that don't get canceled or scrapped do. Avoiding a one-time developmental cost and instead incurring a recurring operational cost makes no sense. In fact, the opposite decision makes much more financial sense. Imagine that your system requires five minutes of downtime on every release. You expect your system to have a five-year life span with monthly releases. (Most companies would like to do more releases per year, but I'm being very conservative.) You can compute the expected cost of downtime, discounted by the time-value of money. It's probably on the order of $1,000,000 (300 minutes of downtime at a very modest cost of $3,000 per minute).

Now suppose you could invest $50,000 to create a build pipeline and deployment process that avoids downtime during releases. That will, at a minimum, avoid the million-dollar loss. It's very likely that it will also allow you to increase deployment frequency and capture market share. But let's stick with the direct gain for now. Most CFOs would not mind authorizing an expenditure that returns 2,000 percent ROI!

Design and architecture decisions are also financial decisions. These choices must be made with an eye toward their implementation cost as well as their downstream costs. The fusion of technical and financial viewpoints is one of the most important recurring themes in this book.