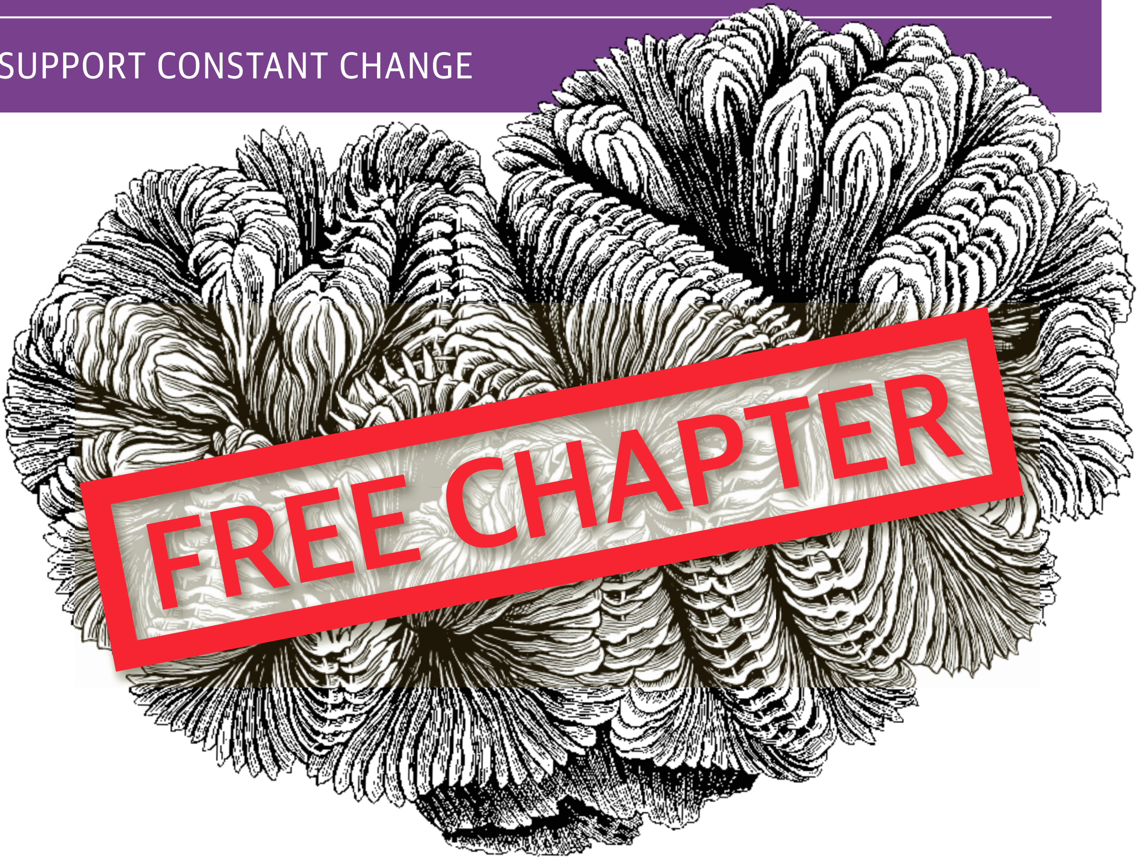
O'REILLY®

Compliments of ThoughtWorks

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE



Neal Ford, Rebecca Parsons & Patrick Kua

Building Evolutionary Architectures

Support Constant Change

Neal Ford, Rebecca Parsons, and Patrick Kua

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Evolutionary Architectures
by Neal Ford, Rebecca Parsons, and Patrick Kua

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Rachel Roumeliotis
Production Editor: Justin Billing
Copyeditor: Christina Edwards
Proofreader: Matthew Burgoyne
Indexer: WordCo, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2017: First Edition

Revision History for the First Edition
2017-09-15: First Release

See *http://www.oreilly.com/catalog/errata.csp?isbn=9781491986363* for release details.

978-1-492-02608-2

[LSI]

Table of Contents

Engineering Incremental Change. 5
Building Blocks 8
Testable 9
Deployment Pipelines 11
Combining Fitness Function Categories 15
Case Study: Architectural Restructuring while Deploying 60 Times/Day 17
Conflicting Goals 19
Case Study: Adding Fitness Functions to PenultimateWidgets' Invoicing Service 20
Hypothesis- and Data-Driven Development 22
Case Study: What to Port? 24

Engineering Incremental Change

An evolutionary architecture supports guided, *incremental* change across multiple dimensions.

—our definition

In 2010, Jez Humble and Dave Farley released *Continuous Delivery*, a collection of practices to enhance the engineering efficiency in software projects. They provided the *mechanism* for building and releasing software via automation and tools but not the *structure* of how to design evolvable software. Evolutionary architecture assumes these engineering practices as prerequisites but addresses how to utilize them to help design evolvable software.

Our definition of evolutionary architecture implies *incremental change*, meaning the architecture should facilitate change in small increments. This chapter describes architectures that support incremental change along with some of the engineering practices used to achieve incremental change, an important building block of evolutionary architecture. We discuss two aspects of incremental change: *development*, which covers how developers build software, and *operational*, which covers how teams deploy software.

Here is an example of the operational side of incremental change. We start with the fleshed out example of incremental change from Chapter 1, which includes additional details about the architecture and deployment environment. PenultimateWidgets, our seller of widgets, has a catalog page backed by a microservice architecture and engineering practices, as illustrated in Figure 1-1.

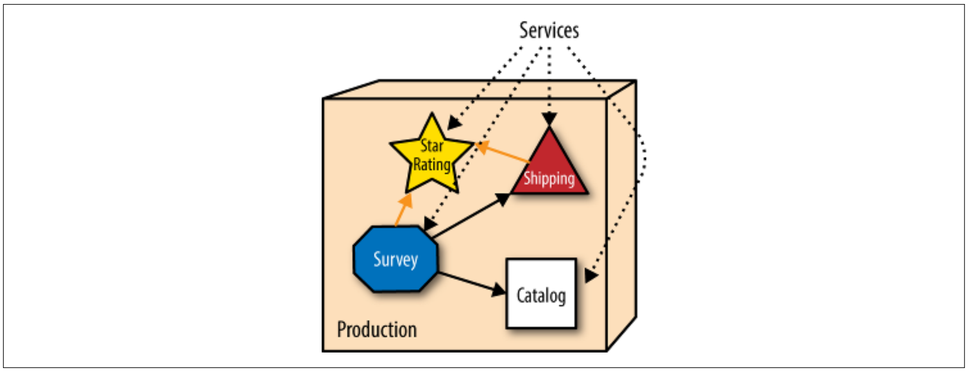


Figure 1-1. Initial configuration of PenultimateWidgets' component deployment

PenultimateWidgets' architects have implemented microservices that are operationally isolated from other services. Microservices implement a *share nothing* architecture: Each service is operationally distinct to eliminate technical coupling and therefore promote change at a granular level. PenultimateWidgets deploys all their services in separate containers to trivialize operational changes.

The website allows users to rate different widgets with star ratings. But other parts of the architecture also need ratings (customer service representatives, shipping provider evaluation, and so on), so they all share the star rating service. One day, the star rating team releases a new version alongside the existing one that allows half-star ratings—a significant upgrade, as shown in Figure 1-2.

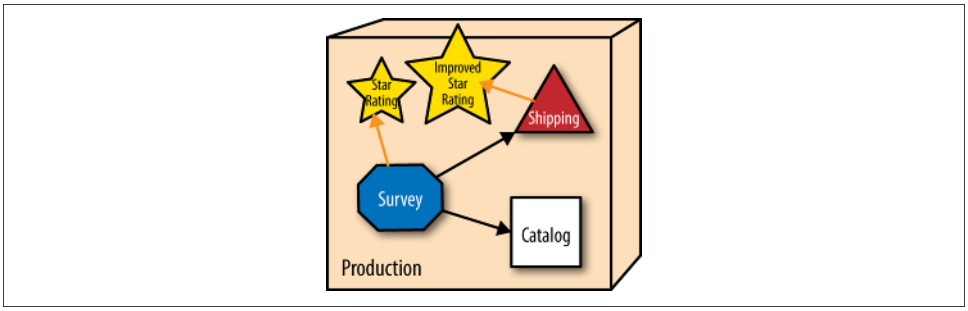


Figure 1-2. Deploying with an improved star rating service showing the addition of the half-star rating

The services that utilize ratings aren't required to migrate to the improved rating service but can gradually transition to the better service when convenient. As time progresses, more parts of the ecosystem that need ratings move to the enhanced version. Part of PenultimateWidgets' DevOps practices include architectural monitoring—monitoring not only the services, but also the routes between services. When the operations group observes that no one has routed to a particular service within a given time interval, they automatically disintegrate that service from the ecosystem, as shown in Figure 1-3.

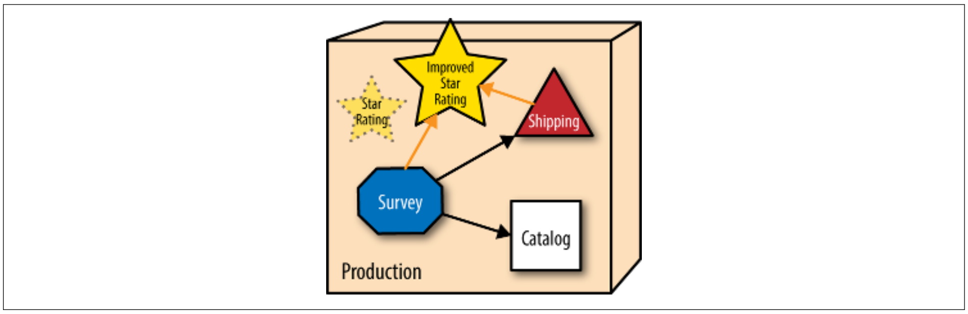


Figure 1-3. All services now use the improved star rating service

The mechanical ability to evolve is one the key components of an evolutionary architecture. Let's dig one level deeper in the abstraction above.

PenultimateWidgets has a fine-grained microservices architecture, where each service is deployed using a container (like Docker) and using a service template to handle infrastructure coupling. Applications within PenultimateWidgets consist of routes between instances of services running—a given service may have multiple instances to handle operational concerns like on-demand scalability. This allows architects to host different versions of services in production and control access via routing. When a deployment pipeline deploys a service, it registers itself (location and contract) with a service discovery tool. When a service needs to find another service, it uses the discovery tool to learn the location and version suitability via the contract.

When the new star rating service is deployed, it registers itself with the service discovery tool and publishes its new contract. The new version of the service supports a broader range of values—specifically, half-point values—than the original. That means the service developers don't have to worry about restricting the supported values. If the new version requires a different contract for callers, it is typical to handle that within the service rather than burden callers with resolving which version to call.

When the team deploys the new service, they don't want to force the calling services to upgrade to the new service immediately. Thus, the architect temporarily changes the star-service endpoint into a proxy that checks to see which version of the service is requested and routes to the requested version. No existing services must change to use the rating service as they always have, but new calls can start taking advantage of the new capability. Old services aren't forced to upgrade and can continue to call the original service as long as they need it. As the calling services decide to use the new behavior, they change the version they request from the endpoint. Over time, the original version falls into disuse, and at some point, the architect can remove the old version from the endpoint when it is no longer needed. Operations is responsible for scanning for services that no other services call anymore (within some reasonable threshold) and garbage collecting the unused services.

All the changes to this architecture, including the provisioning of external components such as the database, happen under the supervision of a deployment pipeline, removing the responsibility of coordinating the disparate moving parts of the deployment from DevOps.

This chapter covers the characteristics, engineering practices, team considerations, and other aspects of building architectures that support incremental change.

Building Blocks

Many of the building blocks required for agility at the architecture level have become mainstream over the last few years under the umbrella of Continuous Delivery and its engineering practices.

Software architects have to determine how systems fit together, often by creating diagrams, with varying degrees of ceremony. Architects often fall into the trap of seeing software architecture as an *equation* they must solve. Much of the commercial tooling sold to software architects reinforces the mathematical illusion of certainty with boxes, lines, and arrows. While useful, these diagrams offer a 2D view—a snapshot of an ideal world—but we live in a 4D world. To flesh out that 2D diagram, we must add specifics. The `ORM` label Figure 1-4 becomes `JDBC 2.1`, evolving into a 3D view of the world, where architects prove their designs in a real production environment using real software. As Figure 1-4 illustrates, over time, changes in business and technology require architects to adopt a 4D view of architecture, making evolution a first-class concern.

Nothing in software is static. Take a computer, for example. Install an operating system and a nontrivial set of software on it, then lock it in a closet for a year. At the end of the year, retrieve it from the closet and plug it into the wall and Internet…and watch it install updates for a long time. Even though no one changed a single bit on the computer, *the entire world kept moving*; this is the dynamic equilibrium we described earlier. Any reasonable architecture plan must include evolutionary change.

When we know how to put architecture into production *and* upgrade it to incorporate inevitable changes (security patches, new versions of software, evolutions of the architecture, and so on) as needed, we've graduated to a 4D world. Architecture isn't a static equation but rather a snapshot of an ongoing process, as illustrated in Figure 1-4.

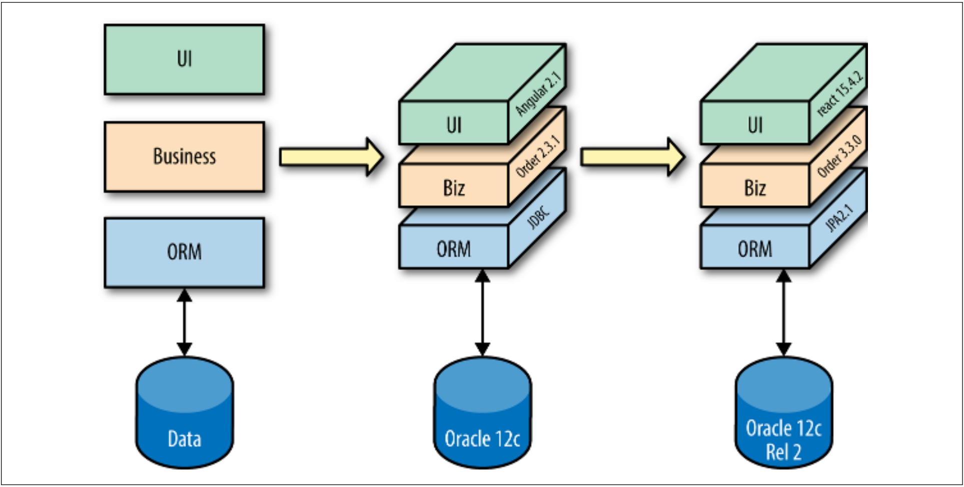


Figure 1-4. Modern architecture must be deployable and changeable to survive the real world

Continuous Delivery and the DevOps movement illustrate the need to implement an architecture and keep it current. There is nothing wrong with modeling architecture and capturing those efforts, but the model is merely the first step.

Architecture is abstract until operationalized, when it becomes a living thing.

Figure 1-4 illustrates the natural evolution of version upgrades and new tool choices. Architectures evolve in other ways as well, as we'll see in the Chapter 6.

Architects cannot judge the long-term viability of any architecture until *design*, *implementation*, *upgrade*, and *inevitable change* are successful. And perhaps even enabled the architecture to withstand unusual occurrences based on incipient unknown unknowns, which we cover in Chapter 6.

Testable

One of the oft ignored "-ilities" of software architecture is *testability*—can characteristics of the architecture submit to automated tests to verify veracity? Unfortunately, it is often difficult to test architecture parts due to lack of tool support.

However, some aspects of an architecture do yield to easy testing. For example, developers can test concrete architectural characteristics like coupling, develop guidelines, and eventually automate those tests.

Here is an example of a fitness function defined at the technical architecture dimension to control the directionality of coupling between components. In the Java ecosystem, JDepend is a metrics tool that analyzes the coupling characteristics of packages. Because JDepend is written in Java, it has an API that developers can leverage to build their own analysis via unit tests.

Consider the fitness function in Example 1-1, expressed as a JUnit test:

Example 1-1. JDepend test to verify the directionality of package imports

```
public void testMatch() {
    DependencyConstraint constraint = new DependencyConstraint();

    JavaPackage persistence = constraint.addPackage("com.xyz.persistence");
    JavaPackage web = constraint.addPackage("com.xyz.web");
    JavaPackage util = constraint.addPackage("com.xyz.util");

    persistence.dependsUpon(util);
    web.dependsUpon(util);

    jdepend.analyze();

    assertEquals("Dependency mismatch",
             true, jdepend.dependencyMatch(constraint));
    }
```

In Example 1-1, we define the packages in our application and then define the rules about imports. One of the bedeviling problems in component-based systems is component cycles—i.e., when component A references component B, which in turn references component A again. If a developer accidentally writes code that imports into `util` from `persistence`, this unit test will fail before the code is committed. We prefer building unit tests to catch architecture violations over using strict development guidelines (with the attendant bureaucratic scolding): It allows developers to focus more on the domain problem and less on plumbing concerns. More importantly, it allows architects to consolidate rules as executable artifacts.

Fitness functions can have any owner, including shared ownership. In the example shown in Example 1-1, the application team may own the directionality fitness function because it is a particular concern for that project. In the same deployment pipeline, fitness functions common across multiple projects may be owned by the security team. In general, the definition and maintenance of fitness functions is a shared responsibility between architects, developers, and any other role concerned with maintaining architectural integrity.

Many things about architecture are testable. Tools exist to test the structural characteristics of architecture such as JDepend (or a similar tool in the .NET ecosystem NDepend). Tools also exist for performance, scalability, resiliency, and a variety of other architectural characteristics. Monitoring and logging tools also qualify: Any tool that helps assess some architectural characteristic qualifies as a fitness function.

Once they have defined fitness functions, architects must ensure that they are evaluated in a timely manner. Automation is the key to continual evaluation. A *deployment pipeline* is often used to evaluate tasks like this. Using a deployment pipeline, architects can define which, when, and how often fitness functions execute.

Deployment Pipelines

Continuous Delivery describes the deployment pipeline mechanism. Similar to a continuous integration server, a deployment pipeline "listens" for changes, then runs a series of verification steps, each with increasing sophistication. Continuous Delivery practices encourage using a deployment pipeline as the mechanism to automate common project tasks, such as testing, machine provisioning, deployments, etc. Open source tools such as GoCD facilitate building these deployment pipelines.

A typical deployment pipeline automatically builds the deployment environment (a container like Docker or a bespoke environment generated by a tool like Puppet or Chef) as shown in Figure 1-5.

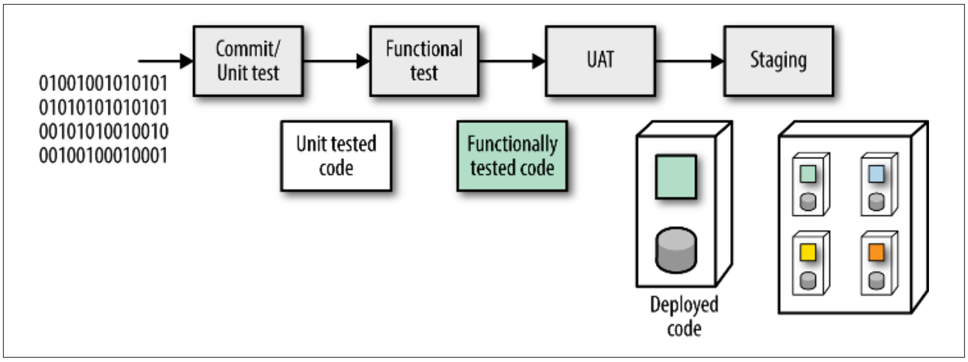


Figure 1-5. Deployment pipeline stages

By building the deployment image that the deployment pipeline executes, developers and operations have a high degree of confidence: The host computer (or virtual machine) is declaratively defined, and it's a common practice to rebuild it from nothing.

Continuous Integration Versus Deployment Pipelines

Continuous integration is a well-known engineering practice in agile projects that encourages developers to integrate as early and as often as possible. To facilitate continuous integration, tools such as ThoughtWorks CruiseControl and other commercial and open source offerings have emerged. Continuous integration provides an "official" build location, and developers enjoy the concept of a single mechanism to ensure working code. However, a continuous integration server also provides a perfect time and place to perform common project tasks such as unit testing, code coverage, metrics, functional testing, and so on. For many projects, the continuous integration server includes a list of tasks to perform whose successful culmination indicates build success. Large projects eventually build an impressive list of tasks.

Deployment pipelines encourage developers to split individual tasks into *stages*. A deployment pipeline includes the concept of multi-stage builds, allowing developers to model as many post-checkin tasks as necessary. This ability to separate tasks discretely supports the broader mandates expected of a deployment pipeline—to verify production readiness—compared to a continuous integration (CI) server primarily focused on integration. Thus, a deployment pipeline commonly includes application testing at multiple levels, automated environment provisioning, and a host of other verification responsibilities.

Some developers try to "get by" with a continuous integration server but soon find they lack the level of separation of tasks and feedback necessary.

The deployment pipeline also offers an ideal way to execute the fitness functions defined for an architecture: It applies arbitrary verification criteria, has multiple stages to incorporate differing levels of abstraction and sophistication of tests, and runs every single time the system changes in any way. A deployment pipeline with fitness functions added is shown in Figure 1-6.

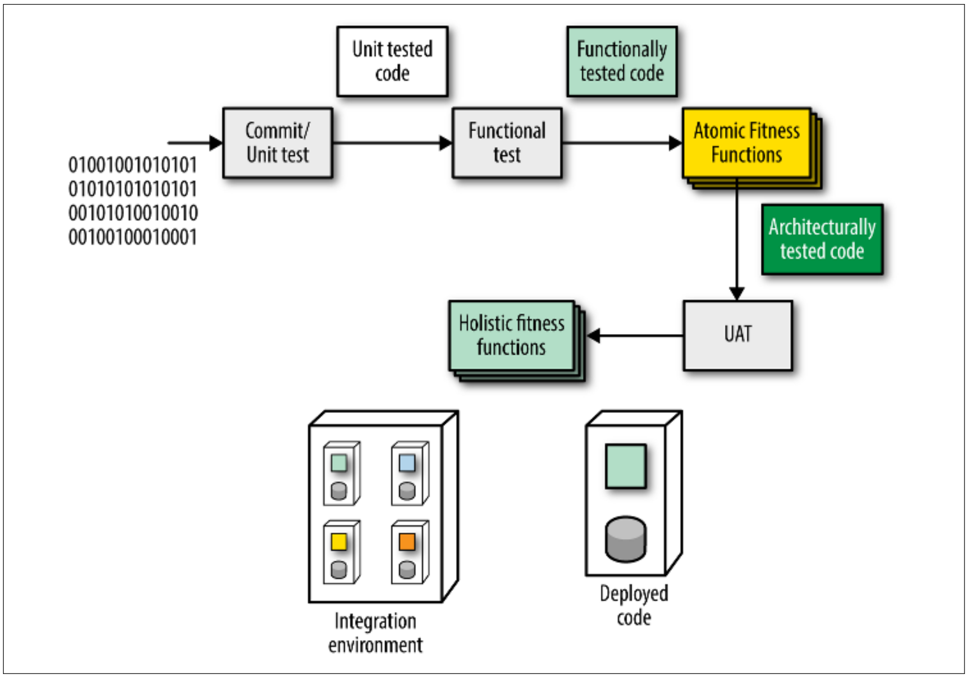


Figure 1-6. A deployment pipeline with fitness functions added as stages

Figure 1-6 shows a collection of atomic and holistic fitness functions with the latter in a more complex integration environment. Deployment pipelines can ensure the rules defined to protect architectural dimensions execute each time the system changes.

PenultimateWidgets Deployment Pipelines

In Chapter 2, we described PenultimateWidgets' spreadsheet of requirements. Once they adopted some of the Continuous Delivery engineering practices, they realized that nonfunctional platform requirements work better in an automated deployment pipeline. To that end, service developers created a deployment pipeline to validate the fitness functions created both by the enterprise architects and by the service team. Now, each time the team makes a change to the service, a barrage of tests validates both the correctness of the code and its overall fitness within the architecture.

Another common practice in evolutionary architecture projects is continuous deployment—using a deployment pipeline to put changes into production contingent on successfully passing the pipeline's gauntlet of tests and other verifications. While continuous deployment is ideal, it requires sophisticated coordination: Developers must ensure changes deployed to production on an ongoing basis don't break things.

To solve this coordination problem, a *fan out* operation is commonly used in deployment pipelines where the pipeline runs several jobs in parallel, as shown in Figure 1-7.

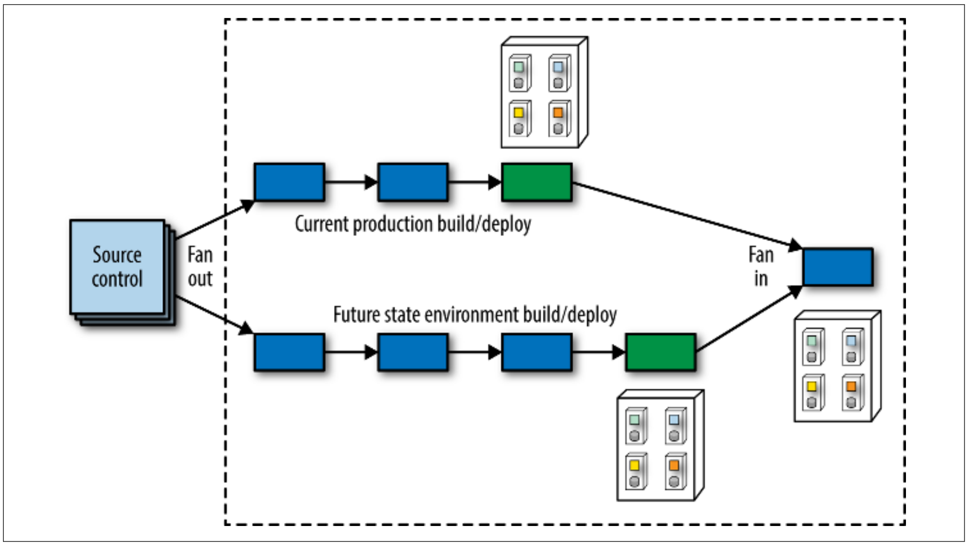

Source control
Fan out
Current production build/deploy
Future state environment build/deploy
Fan in

Figure 1-7. Deployment pipeline fan out to test multiple scenarios

As shown in Figure 1-7, when a team makes a change, they have to verify two things: They haven't negatively affected the current production state (because a successful deployment pipeline execution will deploy code into production) and their changes were successful (affecting the future state environment). A deployment pipeline fan out allows tasks (testing, deploy, and so on) to execute in parallel, saving time. Once the series of concurrent jobs illustrated in Figure 1-7 completes, the pipeline can evaluate the results and if everything is successful, perform a *fan in*, consolidating to a single thread of action to perform tasks like deployment. Note that the deployment pipeline may perform this combination of *fan out* and *fan in* numerous times whenever the team needs to evaluate a change in multiple contexts.

Another common issue with continuous deployment is business impact. Users don't want a barrage of new features showing up on a regular basis but would rather have them staged in a more traditional way such as a "Big Bang" deployment. A common way to accommodate both continuous deployment and staged releases is to use *feature toggles*. By implementing new features hidden underneath feature toggles, developers can safely deploy the feature to production without worrying about users seeing it prematurely.

QA in Production

One beneficial side effect of habitually building new features using feature toggles is the ability to perform QA tasks in production. Many companies don't realize they can use their production environment for exploratory testing. Once a team becomes comfortable using feature toggles, they can deploy those changes to production since most feature toggle frameworks allow developers to route users based on wide variety of criteria (IP address, access control list (ACL), etc.). If a team deploys new features within feature toggles to which only the QA department has access, they can test in production.

Using deployment pipelines in engineering practices, architects can easily apply project fitness functions. Figuring out which stages are needed is a common challenge for developers designing a deployment pipeline. Casting the project's architectural concerns (including evolvability) as fitness functions provides many benefits:

- Fitness functions are designed to have objective, quantifiable results
- Capturing all concerns as fitness function creates a consistent enforcement mechanism
- Having a list of fitness functions allows developers to most easily design deployment pipelines

Determining when in the project's build cycle to run fitness functions, which ones to run, and the proper context is a nontrivial undertaking. However, once the fitness functions inside a deployment pipeline are in place, architects and developers have a high level of confidence that evolutionary changes won't violate the project guidelines. Architectural concerns are often poorly elucidated and sparsely evaluated, often subjectively; creating them as fitness functions allows better rigor and therefore better confidence in the engineering practices.

Combining Fitness Function Categories

Fitness function categories often intersect when implementing them in mechanisms like deployment pipelines. Here are some common mashups of fitness function categories, along with examples.

atomic + triggered

This type of fitness function is exemplified by unit and functional tests run as part of software development. Developers run them to verify changes, and an automation mechanism, such as a deployment pipeline, applies continuous integration to ensure timeliness. A common example of this type of fitness function is a unit test that verifies some aspect of the architectural integrity of the application architecture, such as circular dependencies or cyclomatic complexity.

holistic + triggered
: Holistic, triggered fitness functions are designed to run as part of integration testing via a deployment pipeline. Developers design these tests specifically to test how different aspects of the system interact in well-defined ways. For example, developers may be curious to see what kind of impact tighter security has on scalability. Architects design these tests to intentionally test some integration characteristic in the code base because breakages indicate some architectural shortcoming. Like all triggered tests, developers typically run these fitness functions both during development and as part of a deployment pipeline or continuous integration environment. Generally, these are tests and metrics that have well-known outcomes.

atomic + continual
: Continual tests run as part of the architecture, and developers design around their presence. For example, architects might be concerned that all REST endpoints support the proper verbs, exhibit correct error handling, and support metadata properly and therefore build a tool that runs continually to call REST endpoints (just as normal clients would) to verify the results. The *atomic* scope of these fitness functions suggests that they test just one aspect of the architecture, but *continual* indicates that the tests run as part of the overall system.

holistic + continual
: Holistic, continual fitness functions test multiple parts of the system all the time. Basically, this mechanism represents an agent (or another client) in a system that constantly assesses a combination of architectural and operational qualities. An outstanding example of a real-world continual holistic fitness function is Netflix's Chaos Monkey. When Netflix designed their distributed architecture, they designed it to run on the Amazon Cloud. But engineers were concerned what sort of odd behavior could occur because they have no direct control over their operations, such as high latency, availability, elasticity, and so on, in the Cloud. To assuage their fears, they created Chaos Monkey, eventually followed by an entire open source Simian Army. Chaos Monkey "infiltrates" an Amazon data center and starts making unexpected things happen: Latency goes up, reliability goes down, and other chaos ensues. By designing with Chaos Monkey in mind, each team must build resilient services. The RESTful verification tool mentioned in in the previous section exists as the Conformity Monkey, which checks each service for architect-defined best practices.

Note that Chaos Monkey isn't a testing tool run on a schedule—it runs continuously within Netflix's ecosystem. Not only does this force developers to build systems that withstand problems, it tests the system's validity continually. Having this constant verification built into the architecture has allowed Netflix to build one of the the most robust systems in the world. The Simian Army provides an excellent example of a holistic continual operational fitness function. It runs against multiple parts of the architecture at once, ensuring architectural characteristics (resiliency, scalability, etc.) are maintained.

Holistic, continual fitness functions are the most complex fitness functions for developers to implement but can provide great power, as the following case study illustrates.

Case Study: Architectural Restructuring while Deploying 60 Times/Day

GitHub is a well-known developer-centric website with aggressive engineering practices, deploying on average 60 times a day. They describe a problem in their blog "Move Fast and Fix Things" that will make many architects shudder in horror. It turns out that GitHub has long used a shell script wrapped around command-line Git to handle merges, which works correctly but doesn't scale well enough. The Git engineering team built a replacement library for many command-line Git functions called libgit2 and implemented their merge functionality there, thoroughly testing it locally.

But now they must deploy the new solution into production. This behavior has been part of GitHub since its inception and has worked flawlessly. The last thing the developers want to do is introduce bugs in existing functionality, but they must address technical debt as well.

Fortunately, GitHub developers created and open sourced Scientist, a framework that provides holistic, continual testing to vet changes to code. Example 1-2 gives us the structure of a `Scientist` test.

Example 1-2. Scientist setup for an experiment

```ruby
require "scientist"

class MyWidget
  include Scientist

  def allows?(user)
    science "widget-permissions" do |e|
      e.use { model.check_user(user).valid? } # old way
      e.try { user.can?(:read, model) } # new way
    end # returns the control value
  end
end
```

In Example 1-2, the developer takes the existing behavior and encapsulates it with the `use` block (called the *control*) and adds the experimental behavior to the `try` block (called the *candidate*). The `science` block handles the following details during the invocation of the code:

Decides whether to run the `try` *block*
: Developers configure Scientist to determine how the experiment runs. For example, in this case study—the goal of which was to update their merge functionality—1% of random users tried the new merge functionality. In either case, Scientist *always* returns the results of the `use` block, ensuring the caller always receives the existing behavior in case of differences.

Randomizes the order that `use` *and* `try` *blocks run*
: Scientist does this to prevent accidentally masking bugs due to unknown dependencies. Sometimes the order or other incidental factors can cause false positives; by randomizing their order, the tool makes those faults less likely.

Measures the durations of all behaviors
: Part of Scientist's job is A/B performance testing, so monitoring performance is built in. In fact, developers can use the framework piecemeal—for example, they can use it to measure calls without performing experiments.

Compares the result of `try` *to the result of* `use`
: Because the goal is refactoring existing behavior, Scientist compares and logs the results of each call to see if differences exist.

Swallows (but logs) any exceptions raised in the `try` *block*
: There's always a chance that new code will throw unexpected exceptions. Developers never want end users to see these errors, so the tool makes them invisible to the end user (but logs it for developer analysis).

Publishes all this information
: Scientist makes all its data available in a variety of formats.

For the merge refactoring, the GitHub developers used the following invocation to test the new implementation (called `create_merge_commit_rugged`), as shown in Example 1-3.

Example 1-3. Experimenting with a new merge algorithm

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Merge #{head} into #{base}"
  now = Time.current

  science "create_merge_commit" do |e|
    e.context :base => base.to_s, :head => head.to_s, :repo => repository.nwo
    e.use { create_merge_commit_git(author, now, base, head, commit_message) }
    e.try { create_merge_commit_rugged(author, now, base, head, commit_message) }
  end
end
```

In Example 1-3, the call to `create_merge_commit_rugged` occurred in 1% of invocations, but, as noted in this case study, at GitHub's scale, all edge cases appear quickly.

When this code executes, end users always receive the correct result. If the `try` block returns a different value from `use`, it is logged, and the `use` value is returned. Thus, the worse case for end users is exactly what they would have gotten before the refactoring. After running the experiment for 4 days and experiencing no slow cases or mismatched results for 24 hours, they removed the old merge code and left the new in place.

From our perspective, Scientist is a fitness function. This case study is an outstanding example of the strategic use of a holistic, continous fitness function to allow developers to refactor a critical part of their infrastructure with confidence. They changed a key part of their architecture by running the new version alongside the existing, essentially turning the legacy implementation into a consistency test.

In general, most architectures will have a large number of atomic fitness functions and a few key holistic ones. The determining factor of atomicity comes down to what developers are testing and how broad are the results.

Conflicting Goals

The agile software development process has taught us that the sooner a developer can detect problems, the less effort is required to fix them. One of the side effects of broadly considering all the dimensions in software architecture is the early identification of goals that conflict across dimensions. For example, developers at an organization may want to support the most aggressive pace of change to support new features. Fast change to code implies fast changes to database schemas, but the database administrators are more concerned about stability because they are building a data warehouse. The two evolution goals conflict across the technical and data architecture.

Obviously, some compromise must occur, taking into account the myriad factors that affect the underlying business. Using architecture dimensions as a technique for identifying portions of concern in architecture (plus fitness functions to evaluate them) allows an apples-to-apples comparison, making the prioritization exercise more informed.

Conflicting goals are inevitable. However, discovering and quantifying those conflicts early allows architects to make better informed decisions and create more clearly defined goals and principles.

Case Study: Adding Fitness Functions to PenultimateWidgets' Invoicing Service

Our exemplar company, PenultimateWidgets, has an architecture that includes a service to handle invoicing. The invoicing team wants to replace outdated libraries and approaches but wants to ensure these changes don't impact other teams ability to integrate with them.

The invoicing team identified the following needs:

Scalability
: While performance isn't a big concern for PenultimateWidgets, they handle invoicing details for several resellers, so the invoicing service must maintain availability service-level agreements.

Integration with other services
: Several other services in the PenultimateWidgets ecosystem use invoicing. The team wants to make sure integration points don't break while making internal changes.

Security
: Invoicing means money, and security is always an ongoing concern.

Auditability
: Some state regulations require that changes to taxation code be verified by an independant accountant.

The invoicing team uses a continuous integration server and recently upgraded to on-demand provisioning of the environment that runs their code. To implement evolutionary architecture fitness functions, they implement a deployment pipeline to replace the continuous integration server, allowing them to create several stages of execution, as shown in Figure 1-8.

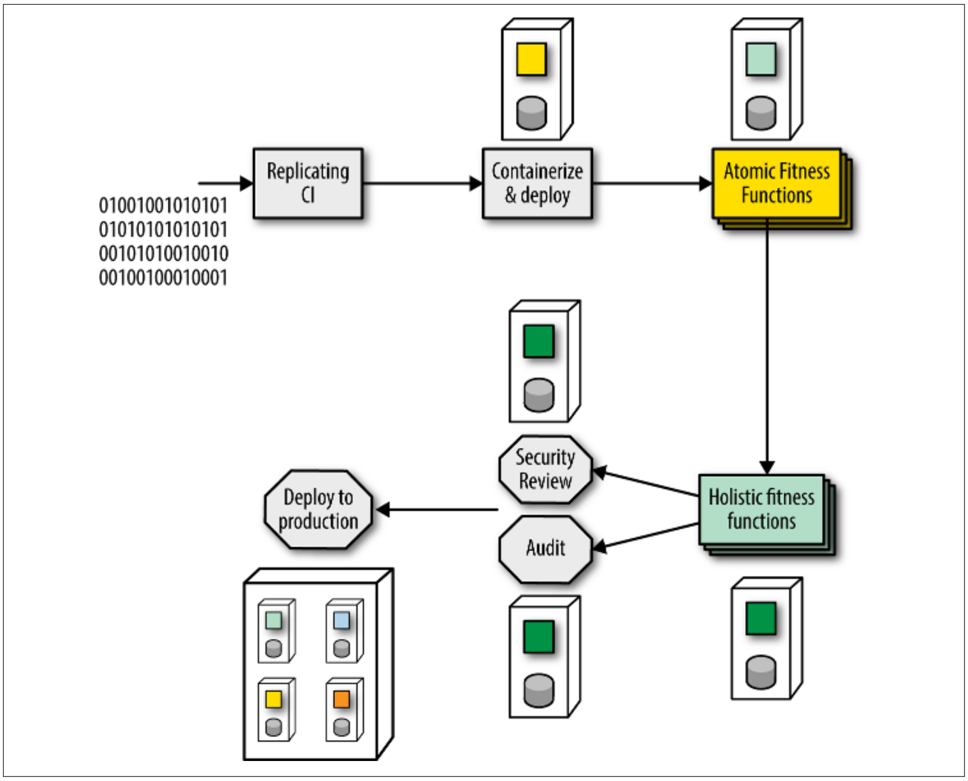


Figure 1-8. PenultimateWidgets deployment pipeline

PenultimateWidgets' deployment pipeline consists of six stages.

Stage 1—Replicating CI
: The first stage replicates the behavior of the former CI server, running unit, and functional tests.

Stage 2—Containerize and Deploy
: Developers use the second stage to build containers for their service, allowing deeper levels of testing, including deploying the containers to a dynamically created test environment.

Stage 3—Atomic Fitness Functions
: In the third stage atomic fitness functions, including automated scalability tests and security penetration testing, are executed. This stage also runs a metrics tool that flags any code within a certain package that developers changed, pertaining to auditability. While this tool doesn't make any deteriminations, it assists a later stage in narrowing in on specific code.

Stage 4—Holistic Fitness Functions
: The fourth stage focuses on holistic fitness functions, including testing contracts to protect integration points and some further scalability tests.

Stage 5a—Security Review (manual)
: This stage includes a manual stage by a specific security group within the organization to review, audit, and assess any security vulnerabilities in the codebase. Deployment pipelines allow the definition of manual stages, triggered on demand by the relevant security specialist.

Stage 5b—Auditing (manual)
: PenultimateWidgets is based in Springfield, where the state mandates specific auditing rules. The invoicing team builds this manual stage into their deployment pipeline, which offers several benefits. First, treating auditing as a fitness function allows developers, architects, auditors, and others to think about this behavior in a unified way—a necessary evaluation to deterimine the system's correct function. Second, adding the evaluation to the deployment pipeline allows developers to assess the engineering impact of this behavior compared equally to other automated evaluations within the deployment pipeline.

  For example, if the security review happens weekly but auditing happens only monthly, the bottleneck to faster releases is clearly the auditing stage. By treating both security and audit as stages in the deployment pipeline, decisions concerning both can be addressed more rationally: Is it worth value to the company to increase release cadence by having consultants perform the necessary audit more often?

Stage 6—Deployment
: The last stage is deployment into the production environment. This is a automated stage for PenultimateWidgets and is triggered only if the two upstream manual stages (*security review* and *audit*) report success.

Interested architects at PenultimateWidgets receive a weekly automatically generated report about the success/failure rate of the fitness functions, helping them gauge health, cadence, and other factors.

Hypothesis- and Data-Driven Development

The GitHub example in "Case Study: Architectural Restructuring while Deploying 60 Times/Day" on page 17 using the `Scientist` framework is an example of *data-driven development*—allow data to drive changes and focus efforts on technical change. A similar approach that incorporates the business rather than technical concerns is *hypothesis-driven development*.

In the week between Christmas 2013 and New Year's Day 2014, Facebook encountered a problem: More photos were uploaded to Facebook in that week than all the photos on Flickr, and more than a million of them were flagged as offensive. Facebook allows users to flag photos they believe potentially offensive and then reviews them to determine objectively if they are. But this dramatic increase in photos created a problem: There was not enough staff to review the photos.

Fortunately, Facebook has modern DevOps and the ability to perform experiments on their users. When asked about the chances a typical Facebook user has been involved in an experiment, one Facebook engineer claimed "Oh, one hundred percent —we routinely have more than twenty experiments running at time." They used this experimental capability to ask users follow-up on questions about *why* photos were deemed offensive and discovered many delightful quirks of human behavior. For example, people don't like to admit that they look bad in a photo but will freely admit that the photographer did a poor job. By experimenting with different phrasing and questions, the engineers could query their actual users to determine why they flagged a photo as offensive. In a relatively short amount of time, Facebook shaved off enough false positives to restore offensive photos to a manageable problem by building a platform that allowed for experimentation.

In the book *Lean Enterprise* (O'Reilly, 2014), Barry O'Reilly describes the modern process of *hypothesis-driven development*. Under this process, rather than gathering formal requirements and spending time and resources building features into applications, teams should leverage the scientific method instead. Once teams have created the minimal viable product version of an application (whether as a new product or by performing maintenance work on an existing application), they can build hypotheses during new feature ideation rather than requirements. Hypothesis-driven devlopment hypotheses are couched in terms of the hypothesis to test, what experiments can determine the results, and what validating the hypothesis means to future application development.

For example, rather than change the image size for sales items on a catalog page because a business analyst thought it was a good idea, state it as a hypothesis instead: If we make the sales images bigger, we hypothesize that it will lead to a 5% increase in sales for those items. Once the hypothesis is in place, run experiments via A/B testing —one group with bigger sales images and one without—and tally the results.

Even agile projects with engaged business users incrementally build themselves into a bad spot. An individual decision by a business analyst may make sense in isolation, but when combined with other features may ultimately degrade the overall experience. In an excellent case study, mobile.de followed a logical path of accruing new features haphazardly to the point where sales were diminishing, at least in part because their UI had become so convoluted, as is often the result of development continuing on mature software products. Several different philosophical approaches were: more listings, better prioritization, or better grouping. To help them make this decision, they built three versions of the UI and allowed their users to decide.

The engine that drives agile software methodologies is the nested feedback loop: testing, continuous integration, iterations, etc. And yet, the part of the feedback loop that incorporates the ultimate users of the application has eluded teams. Using hypothesis-driven development, we can incorporate users in an unprecedented way, learning from behavior and building what users really find valuable.

Hypothesis-driven development requires the coordination of many moving parts: evolutionary architecture, modern DevOps, modified requirements gathering, and the ability to run multiple versions of an application simultaneously. Service-based architectures (like microservices) usually achieve side-by-side versions by intelligent routing of services. For example, one user may execute the application using a particular constellation of services while another request may use an entirely different set of instances of the same services. If most services include many running instances (for scalability, for example), it becomes trivial to make some of those instances slightly different with enhanced functionality, and to route some users to those features.

Experiments should run long enough to yield significant results. Generally, it is preferable to find a measurable way to determine better outcomes rather than annoy users with things like pop-up surveys. For example, does one hypothesized workflow allow the user to complete a task with fewer keystrokes and clicks? By silently incorporating users into the development and design feedback loop, you can build much more functional software.

Case Study: What to Port?

One particular PenultimateWidgets application has been a workhorse, developed as a Java Swing application over the better part of a decade and continually growing new features. The company decided to port it to the web application. However, now the business analysts face a difficult decision: How much of the existing sprawling functionality should they port? And, more practically, what order should they implement the ported features of the new application to deliver the most functionality quickly?

One of the architects at PenultimateWidgets asked the business analysts what the most popular features were, and they had no idea! Even though they have been specifying the details of the application for years, they had no real understanding of how users used the application. To learn from users, the developers released a new version of the legacy application with logging enabled to track which menu features users actually used.

After a few weeks, they harvested the results, providing an excellent road map of what features to port and in what order. They discovered that the invoicing and customer lookup features were most commonly used. Surprisingly, one subsection of the application that had taken great effort to build had very little use, leading the team to decide to leave that functionality out of the new web application.

About the Authors

Neal Ford is Director, Software Architect, and Meme Wrangler at ThoughtWorks, a software company and a community of passionate, purpose-led individuals who think disruptively to deliver technology to address the toughest challenges, all while seeking to revolutionize the IT industry and create positive social change. Before joining ThoughtWorks, Neal was the Chief Technology Officer at The DSW Group, Ltd., a nationally recognized training and development firm.

Neal has a degree in Computer Science from Georgia State University specializing in languages and compilers and a minor in mathematics specializing in statistical analysis. He is an internationally recognized expert on software development and delivery, especially in the intersection of agile engineering techniques and software architecture. Neal has authored magazine articles, seven books (and counting), and dozens of video presentations and has spoken at hundreds of developers conferences worldwide. The topics of these works include software architecture, continuous delivery, functional programming, and cutting edge software innovations, as well as a business-focused book and video in improving technical presentations. His primary consulting focus is the design and construction of large-scale enterprise applications. If you have an insatiable curiosity about Neal, visit his web site at nealford.com.

Dr. Rebecca Parsons is ThoughtWorks' Chief Technology Officer with decades-long applications development experience across a range of industries and systems. Her technical experience includes leading the creation of large-scale distributed object applications, the integration of disparate systems, and working with architecture teams. Separate from her passion for deep technology, Dr. Parsons is a strong advocate for diversity in the technology industry.

Before coming to ThoughtWorks, Dr. Parsons worked as an assistant professor of computer science at the University of Central Florida where she taught courses in compilers, program optimization, distributed computation, programming languages, theory of computation, machine learning, and computational biology. She also worked as a Director's Postdoctoral Fellow at the Los Alamos National Laboratory researching issues in parallel and distributed computation, genetic algorithms, computational biology, and nonlinear dynamical systems.

Dr. Parsons received a Bachelor of Science degree in Computer Science and Economics from Bradley University, a Master's of Science in Computer Science from Rice University, and her Ph.D. in Computer Science from Rice University. She is also the co-author of *Domain-Specific Languages*, *The ThoughtWorks Anthology*, and *Building Evolutionary Architectures*.

Patrick Kua is a Principal Technical Consultant at ThoughtWorks, having worked in the technology industry for over 15 years. He is well known for bringing a balanced blend between technology, people, and process to improve the effectiveness of software delivery. You can also find him speaking at many conferences on the topics of technical leadership, architecture, and building strong engineering cultures.

He is author of *The Retrospective Handbook: A Guide for Agile Teams and Talking with Tech Leads: From Novices to Practitioners* and established a regular training program to support developers transitioning into the role of a Tech Lead and/or Architect.

You can discover more about him at his website, *thekua.com* or reach out to him on twitter at @patkua

Colophon

The animal on the cover of *Building Evolutionary Architectures* is the open brain coral (*Trachyphyllia geoffroyi*). Also known as a "folded brain" or "crater" coral, this large-polyp stony (LPS) coral is native to the Indian Ocean. Known for its distinctive folds, bright colors, and hardiness, this free-living coral subsists on the photosynthetic output of a surface layer of zooxanthellae during the day, while at night it extends tentacles from its polyps to steer prey, which include various plankton as well as small fish, into one of its mouths (some open brain corals have two or three of them).

Because of its striking appearance and easy-to-accomodate diet, *Trachyphyllia geoffroyi* is a popular choice for aquariums, where it thrives in the bottom layer of sand and/or silt resembling the shallow seafloors of its native habitat. They benefit from an environment with moderate water flow and rich with plant and animal matter to consume.

Trachyphyllia geoffroyi is listed on the IUCN Red List at Near Threatened status. Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Jean Vincent Félix Lamouroux's *Exposition Methodique des genres de L'Ordre des Polypiers*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.